2017 IEEE/ACM 39th International Conference on Software Engineering

Learning Syntactic Program Transformations from Examples

Reudismam Rolim*, Gustavo Soares*†, Loris D'Antoni‡,
Oleksandr Polozov§, Sumit Gulwani¶, Rohit Gheyi*, Ryo Suzuki‖, Björn Hartmann†
*UFCG, Brazil, †UC Berkeley, USA, ‡University of Wisconsin-Madison, USA
§University of Washington, USA, ¶Microsoft, USA, ‖University of Colorado Boulder, USA
reudismam@copin.ufcg.edu.br, gsoares@dsc.ufcg.edu.br, loris@cs.wisc.edu, polozov@cs.washington.edu,
sumitg@microsoft.com, rohit@dsc.ufcg.edu.br, ryo.suzuki@colorado.edu, bjoern@eecs.berkeley.edu

***Abstract*—Automatic program transformation tools can be valuable for programmers to help them with refactoring tasks, and for Computer Science students in the form of tutoring systems that suggest repairs to programming assignments. However, manually creating catalogs of transformations is complex and time-consuming. In this paper, we present REFAZER, a technique for automatically learning program transformations. REFAZER builds on the observation that code edits performed by developers can be used as input-output examples for learning program transformations. Example edits may share the same structure but involve different variables and subexpressions, which must be generalized in a transformation at the right level of abstraction. To learn transformations, REFAZER leverages state-of-the-art programming-by-example methodology using the following key components: (a) a novel domain-specific language (DSL) for describing program transformations, (b) domain-specific deductive algorithms for efficiently synthesizing transformations in the DSL, and (c) functions for ranking the synthesized transformations.**

We instantiate and evaluate REFAZER in two domains. First, given examples of code edits used by students to fix incorrect programming assignment submissions, we learn program transformations that can fix other students' submissions with similar faults. In our evaluation conducted on 4 programming tasks performed by 720 students, our technique helped to fix incorrect submissions for 87% of the students. In the second domain, we use repetitive code edits applied by developers to the same project to synthesize a program transformation that applies these edits to other locations in the code. In our evaluation conducted on 56 scenarios of repetitive edits taken from three large C# open-source projects, REFAZER learns the intended program transformation in 84% of the cases using only 2.9 examples on average.

***Keywords*—Program transformation, program synthesis, tutoring systems, refactoring.**

I. INTRODUCTION

As software evolves, developers edit program source code to add features, fix bugs, or refactor it. Many such *edits* have already been performed in the past by the same developers in a different codebase location, or by other developers in a different program/codebase. For instance, to apply an API update, a developer needs to locate all references to the old API and consistently replace them with the new API [1, 2]. As another example, in programming courses student submissions that exhibit the same fault often need similar fixes. For large classes such as *massive open online courses* (MOOCs), manually providing feedback to different students is an unfeasible burden on the teaching staff.

Since applying repetitive edits manually is tedious and error-prone, developers often strive to automate them. The space of tools for automation of repetitive code edits contains Integrated Development Environments (IDEs), static analyzers, and various domain-specific engines. IDEs, such as Visual Studio [3] or Eclipse [4], include features that automate some code *transformations*, such as adding boilerplate code (e.g., equality comparisons) and code refactoring (e.g., *Rename*, *Extract Method*). Static analyzers, such as ReSharper [5], Coverity [6], ErrorProne [7], and Clang-tidy [8] automate removal of suspicious code patterns, potential bugs, and verbose code fragments. In an education context, AutoGrader [9] uses a set of transformations provided by an instructor to fix common faults in introductory programming assignments.

All aforementioned tool families rely on predefined catalogs of recognized transformation classes, which are hard to extend. These limitations inspire a natural question:

Is it possible to learn program transformations from examples?

Our key observation is that code edits gathered from repositories and version control history constitute *input-output examples* for learning program transformations.

The main challenge of example-based learning lies in abstracting concrete code edits into classes of *transformations* representing these edits. For instance, Fig. 1 shows similar edits performed by different students to fix the same fault in their submissions for a programming assignment. Although the edits share some structure, they involve different expressions and variables. Therefore, a transformation should partially abstract these edits as in Fig. 1d.

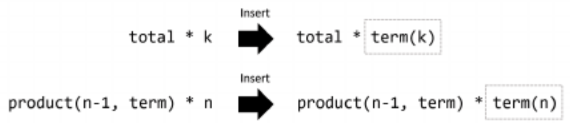
However, examples are highly ambiguous, and many different transformations may satisfy them. For instance, replacing `<name>` by `<exp>` in the transformation will still satisfy the examples in Fig. 1. In general, learning either the most specific or the most general transformation is undesirable,

```
  def product(n, term):
    total, k = 1, 1
    while k<=n:
-     total = total*k
+     total = total*term(k)
      k = k+1
    return total
```

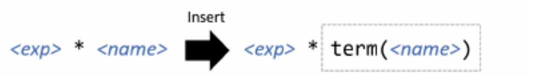
(a) An edit applied by a student to fix the program.

```
  def product(n, term):
    if (n==1):
      return 1
-   return product(n-1, term)*n
+   return product(n-1, term)*term(n)
```

(b) An edit applied by another student fixing the same fault.



(c) Similar tree edits applied in (a) and (b), respectively. Each edit inserts a concrete subtree to the right-hand side of the $*$ operator. The two edits share the same structure but involve different variables and expressions.



(d) A rewrite rule that captures the two edits in (a) and (b).

Fig. 1: An example of a common fault made by different students, two similar edits that can fix different programs, and a program transformation that captures both edits.

as they are likely to respectively produce false negative or false positive edits on unseen programs. Thus, we need to **(a)** learn and store a *set* of consistent transformations efficiently, and **(b)** rank them with respect to their trade-offs between over-generalization and over-specialization. To resolve these challenges, we leverage state-of-the-art software engineering research to learn such transformations automatically using *Inductive Programming* (IP), or *Programming-by-Example* (PBE) [10], which has been successfully applied to many domains, such as text transformation [11], data cleaning [12], and layout transformation [13].

Our technique In this paper, we propose REFAZER,[1] an IP technique for synthesizing program transformations from examples. REFAZER is based on the PROSE [14] Inductive Programming framework. We specify a *domain-specific language* (DSL) that describes a rich space of program transformations that commonly occur in practice. In our DSL, a program transformation is defined as a sequence of distinct *rewrite rules* applied to the *abstract syntax tree* (AST). Each rewrite rule matches some subtrees of the given AST and outputs modified versions of these subtrees. Additionally, we specify constraints for our DSL operators based on the input-output examples to reduce the search space of transformations, allowing PROSE to efficiently synthesize them. Finally, we define functions to rank the synthesized transformations based on their DSL structure.

[1]http://www.dsc.ufcg.edu.br/~spg/refazer/

Evaluation We evaluated REFAZER in two domains: learning transformations to fix submissions for introductory programming assignments and learning transformations to apply repetitive edits to large codebases.

Our first experiment is motivated by large student enrollments in CS courses, e.g., in MOOCs, where automatically grading student submissions and providing personalized feedback is challenging. In this experiment, we mine submissions to programming assignments to collect examples of edits applied by students to fix their code. We then use these examples to synthesize transformations, and we try using the learned transformations to fix any new students' submissions with similar types of faults. We say that a submission is "fixed" if it passes the set of tests provided by course instructors. Synthesized fixes can then be used for grading or turned into hints to help students locate faults and correct misconceptions. In our evaluation conducted on 4 programming tasks performed by 720 students, REFAZER synthesizes transformations that fix incorrect submissions for 87% of the students.

Our second experiment is motivated by the fact that certain repetitive tasks occurring during software evolution, such as complex forms of code refactoring, are beyond the capabilities of current IDEs and have to be performed manually [15, 16]. In this experiment, we use repetitive code edits applied by developers to the same project to synthesize a program transformation that can be applied to other locations in the code. We performed a study on three popular open-source C# projects (Roslyn [17], Entity Framework [18], and NuGet [19]) to identify and characterize repetitive code transformations. In our evaluation conducted on 56 scenarios of repetitive edits, REFAZER learns the intended program transformation in 84% of the cases using 2.9 examples on average. The learned transformations are applied to as many as 60 program locations. Moreover, in 16 cases REFAZER synthesized transformations on more program locations than the ones present in our dataset, thus suggesting potentially missed locations to the developers.

Contributions This paper makes the following contributions:

- REFAZER, a novel technique that leverages state-of-the-art IP methodology to efficiently solve the problem of synthesizing transformations from examples (Section III);
- An evaluation of REFAZER in the context of learning fixes for students' submissions to introductory programming assignments (Section IV-A);
- An evaluation of REFAZER in the context of learning transformations to apply repetitive edits in open-source industrial C# code (Section IV-B).

II. MOTIVATING EXAMPLES

We start by describing two motivating examples of repetitive program transformations.

A. *Fixing programming assignment submissions*

Assignments in introductory programming courses are often graded using a test suite, provided by the instructors. However, many students struggle to understand the fault in their code when a test fails. To provide more detailed feedback (e.g., fault location or its description), teachers typically compile a *rubric*

```
-  while (receiver.CSharpKind() == SyntaxKind.ParenthesizedExpression)
+  while (receiver.IsKind(SyntaxKind.ParenthesizedExpression))

-  foreach (var m in modifiers) {if (m.CSharpKind() == modifier) return true; };
+  foreach (var m in modifiers) {if (m.IsKind(modifier)) return true; };
```

Fig. 2: Repetitive edits applied to the Roslyn source code to perform a refactoring.

of common types of faults and detect them with simple checks. With a large variety of possible faults, manually implementing these checks can be laborious and error-prone.

However, many faults are common and exhibit themselves in numerous unrelated student submissions. Consider the Python code in Fig. 1a. It describes two submission attempts to solve a programming assignment in the course "The Structure and Interpretation of Computer Programs" (CS61A) at UC Berkeley,[2] an introductory programming class with more than 1,500 enrolled students. In this assignment, the student is asked to write a program that computes the product of the first `n` terms, where `term` is a function. The original code, which includes line 4 instead of line 5, is an incorrect submission for this assignment, and the subsequent student submission fixes it by replacing line 4 with line 5. Notably, the fault illustrated in Fig. 1 was a common fault affecting more than 100 students in the Spring semester of 2016 and Fig. 1b shows a recursive algorithm proposed by a different student with the same fault.

To alleviate the burden of compiling manual feedback, we propose to automatically learn the rubric checks from the student submissions. Existing tools for such automatic learning [1 , 20] cannot generate a transformation that is general enough to represent both the edits shown in Fig. 1c due to their limited forms of abstraction. In REFAZER, this transformation is described as a rewrite rule shown in Fig. 1d. This rewrite rule pattern matches any subtree of the program's AST whose root is a $\star$ operation with a variable as the second operand and inserts a `term` application on top of that variable. Notice that the rewrite rule abstracts both the variable name and the first operand of the $\star$ operator.

B. *Repetitive codebase edits*

We now illustrate how REFAZER automates repetitive codebase editing. The following example is found in Roslyn, Microsoft's library for compilation and code analysis for C# and VB.NET. Consider the edits shown in Fig. 2, where, for every comparison instance with an object returned by the method `CSharpKind`, the developer replaces the == operator with an invocation of the new method `IsKind`, and passes the right-hand side expression as the method's argument. Such refactoring is beyond abilities of IDEs due to its context sensitivity. In contrast, REFAZER generalizes the two example edits in Fig. 2 to the intended program transformation, which can be applied to all other matching AST subtrees in the code.

When we analyzed the commit *8c14644*[3] in the Roslyn repository, we observed that the developer applied this edit to 26 locations in the source code. However, the transformation generated by REFAZER applied this edit to 718 more locations. After we presented the results to the Roslyn developers, they confirmed that the locations discovered by REFAZER should have been covered in the original commit.

[2]http://cs61a.org/

[3]https://github.com/dotnet/roslyn/commit/8c14644

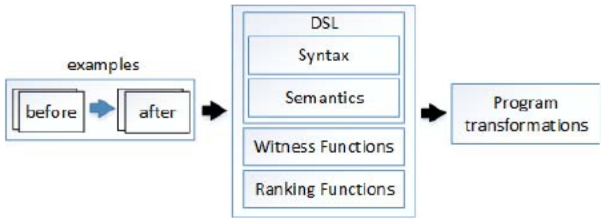


Fig. 3: The workflow of REFAZER. It receives an example-based specification of edits as input and returns a set of transformations that satisfy the examples.

III. TECHNIQUE

In this section, we describe our technique for synthesizing program transformations from input-output examples. REFAZER builds on PROSE [14], a framework for program synthesis from examples and under-specifications.

In PROSE, an application designer defines a *domain-specific language* (DSL) for the desired tasks. The synthesis problem is given by a *spec* φ, which contains a set of program inputs and constraints on the desired program's outputs on these inputs (e.g., examples of these outputs). PROSE synthesizes a set of programs in the DSL that is consistent with φ, using a combination of *deduction*, *search*, and *ranking*:

- Deduction is a top-down walk over the DSL grammar, which iteratively *backpropagates* the spec φ on the desired program to necessary specs on the subexpressions of this program. In other words, it reduces the synthesis problem to smaller synthesis subproblems using a divide-and-conquer dynamic programming algorithm over the desired program's structure.
- Search is an enumerative algorithm, which iteratively constructs candidate subexpressions in the grammar and verifies them for compliance with the spec φ [21].
- Ranking consists of picking the most robust program from synthesized set of programs consistent with φ. Because examples are highly ambiguous, this set may contain up to 10^{20} programs [14], and quickly eliminating undesirable candidates is paramount for a user-friendly experience.

REFAZER consists of three main components, which are illustrated in Fig. 3:

- *A DSL for describing program transformations*. Its operators partially abstract the edits provided as examples. The DSL is expressive enough to capture common transformations but restrictive enough to allow efficient synthesis.
- *Witness functions*. In PROSE, a *witness function* ω_F is a backpropagation procedure, which, given a spec φ on a desired program on kind $F(e)$, deduces a necessary (or even sufficient) spec $\varphi_e = \omega_F(\varphi)$ on its subexpression e.

$$
\begin{array}{lcl}
\textit{transformation} & ::= & \texttt{Transformation}(\textit{rule}_1, \ldots, \textit{rule}_n) \\
\textit{rule} & ::= & \texttt{Map}(\lambda x \rightarrow \textit{operation}, \textit{locations}) \\
\textit{locations} & ::= & \texttt{Filter}(\lambda x \rightarrow \texttt{Match}(x, \textit{match}), \texttt{AllNodes}()) \\
\textit{match} & ::= & \texttt{Context}(\textit{pattern}, \textit{path}) \\
\textit{pattern} & ::= & \textit{token} \mid \texttt{Pattern}(\textit{token}, \textit{pattern}_1, \ldots, \textit{pattern}_n) \\
\textit{token} & ::= & \texttt{Concrete}(\textit{kind}, \textit{value}) \mid \texttt{Abstract}(\textit{kind}) \\
\textit{path} & ::= & \texttt{Absolute}(s) \mid \texttt{Relative}(\textit{token}, k) \\
\textit{operation} & ::= & \texttt{Insert}(x, \textit{ast}, k) \mid \texttt{Delete}(x, \textit{ref}) \\
 & \mid & \texttt{Update}(x, \textit{ast}) \quad \mid \texttt{Prepend}(x, \textit{ast}) \\
\textit{ast} & ::= & \textit{const} \mid \textit{ref} \\
\textit{const} & ::= & \texttt{ConstNode}(\textit{kind}, \textit{value}, \textit{ast}_1, \ldots, \textit{ast}_n) \\
\textit{ref} & ::= & \texttt{Reference}(x, \textit{match}, k)
\end{array}
$$

Fig. 4: A core DSL $\mathcal{L}_{\text{T}}$ for describing AST transformations. $kind$ ranges over possible AST kinds of the underlying programming language, and $value$ ranges over all possible ASTs. s and k range over strings and integers, respectively.

Witness functions enable efficient top-down synthesis algorithms in PROSE.[4]

- *Ranking functions*. Since example-based specifications are incomplete, the synthesized abstract transformation may not perform the desired transformation on other input programs. We specify *ranking functions* that rank a transformation based on its robustness (i.e., likelihood of it being correct in general).

A. A DSL for AST transformations

In this section, we present our DSL for program transformations, hereinafter denoted $\mathcal{L}_{\text{T}}$. It is based on tree edit operators (e.g., `Insert`, `Delete`, `Update`), list processing operators (`Filter`, `Map`), and pattern-matching operators on trees. The syntax of $\mathcal{L}_{\text{T}}$ is formally given in Fig. 4.

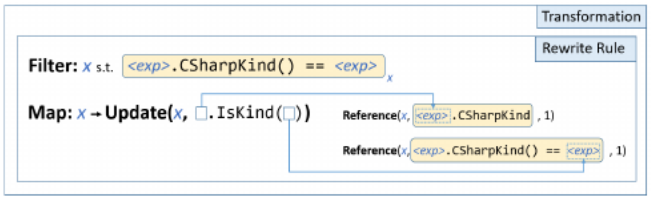
A *transformation* T on an AST is a list of *rewrite rules* (or simply "rules") $r_1, \ldots, r_n$. Each rule r_i specifies an *operation* O_i that should be applied to some set of *locations* in the input AST. The locations are chosen by *filtering* all nodes within the input AST w.r.t. a *pattern-matching predicate*.

Given an input AST P, each rewrite rule r produces a *list of concrete edits* that may be applied to the AST. Each such edit is a replacement of some node in P with a new node. This set of edits is typically an overapproximation of the desired transformation result on the AST; the precise method for applying the edits is domain-specific (e.g., based on verification via unit testing). We discuss the application procedures for our studied domains in Section IV. In the rest of this subsection, we focus on the semantics of the rules that suggest the edits.

A rewrite rule consists of two parts: a *location expression* and an *operation*. A location expression is a `Filter` operator on a set of sub-nodes of a given AST. Its predicate $\lambda x \rightarrow \texttt{Match}(x, \texttt{Context}(pattern, path))$ matches each sub-node x with a *pattern expression*.

Patterns A pattern expression $\texttt{Context}(pattern, path)$ checks the *context* of the node x against a given $pattern$.

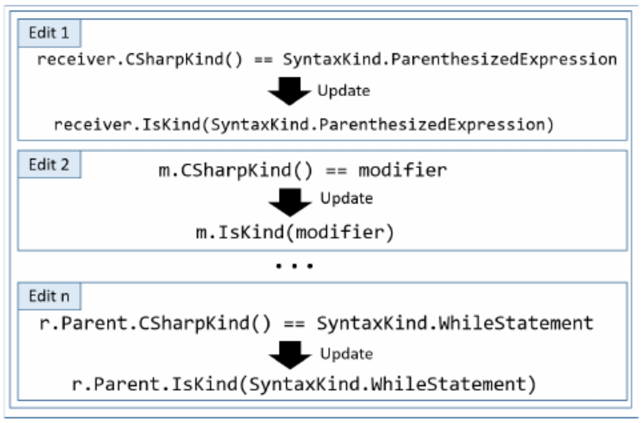
[4]Another view on witness functions ω_F is that they simply implement *inverse semantics* of F, or a generalization of inverse semantics w.r.t. some *constraints* on the output of F instead of just its *value*.


Transformation
Rewrite Rule
Filter: x s.t. <exp>.CSharpKind() == <exp>
x
Map: x → Update(x, □.IsKind(□))
Reference(x, <exp>.CSharpKind , 1)
Reference(x, <exp>.CSharpKind() == <exp> , 1)

(a) A synthesized AST transformation.

```
while (receiver.CSharpKind() ==
  SyntaxKind.ParenthesizedExpression) {...} ...
foreach (var m in modifiers) {
  if (m.CSharpKind() == modifier) {return true;}
}; ...
if (r.Parent.CSharpKind() ==
  SyntaxKind.WhileStatement) {...}
```

(b) A C# program used as an input to the transformation.



```
receiver.CSharpKind() == SyntaxKind.ParenthesizedExpression
receiver.IsKind(SyntaxKind.ParenthesizedExpression)
```

```
m.CSharpKind() == modifier
m.IsKind(modifier)
```

```
r.Parent.CSharpKind() == SyntaxKind.WhileStatement
r.Parent.IsKind(SyntaxKind.WhileStatement)
```

(c) A list of edits produced after instantiating (a) to (b).

Fig. 5: An example of a synthesized transformation and its application to a C# program, which results in a list of edits.

Here $pattern$ is a combination of `Concrete` tokens (which match a concrete AST) and `Abstract` tokens (which match only the AST kind, such as `IfStatement`). In addition, a *path expression* specifies the expected position of x in the context that is described by $pattern$, using a notation similar to XPath [22]. This allows for a rich variety of possible pattern-matching expressions, constraining the ancestors or the descendants of the desired locations in the input AST.

Example 1. Fig. 5 shows a transformation that describes our running example from Fig. 2. This transformation contains one rewrite rule. Its location expression is

$$\texttt{Filter}(\lambda x \rightarrow \texttt{Context}(\pi, \texttt{Absolute}("")))$$

where

$$
\begin{aligned}
\pi &= \texttt{Pattern}(\boxed{\texttt{==}}, \texttt{Pattern}(\boxed{\texttt{.}}, t_e, t_m), t_e) \\
t_e &= \texttt{Abstract}(\boxed{\texttt{<exp>}}) \\
t_m &= \texttt{Concrete}(\boxed{\texttt{<call>}}, \texttt{"CSharpKind()"})
\end{aligned}
$$

The path expression `Absolute("")` specifies that the expected position of a location x in π should be at the root – that is, the pattern π should match the node x itself.

Operations Given a list of locations selected by the `Filter` operator, a rewrite rule applies an *operation* to each of them. An operation O takes as input an AST x and performs one of the standard tree edit procedures [23, 24] on it:

- `Insert` some fresh AST as the k^{th} child of x;
- `Delete` some sub-node from x;
- `Update` x with some fresh AST;
- `Prepend` some fresh AST as the preceding sibling of x.

An operation creates fresh ASTs using a combination of *constant ASTs* `ConstNode` and *reference ASTs* `Reference`, extracted from the location node x. Reference extraction uses the same pattern-matching language, described above. In particular, it can match over the ancestors or descendants of the desired reference. Thus, the semantics of reference extraction `Reference`(x, `Context`($pattern$, $path$), k) is:

1) Find all nodes in x s.t. their surrounding context matches $pattern$, and they are located at $path$ within that context;
2) Out of all such nodes, extract the k^{th} one.

Example 2. For our running example from Fig. 2, the desired rewrite rule applies the following operation to all nodes selected by the location expression from Example 1:

$$\texttt{Update}(x, \texttt{ConstNode}(\boxed{\texttt{<call>}}, \texttt{"IsKind"}, \ell_1, \ell_2))$$

where

$$\begin{aligned}
\ell_1 &= \texttt{Reference}(x, \texttt{Context}(\pi_1, s_1), 1)\\
\ell_2 &= \texttt{Reference}(x, \texttt{Context}(\pi_2, s_2), 1)\\
\pi_1 &= \texttt{Pattern}(\boxed{\texttt{.}}, t_e, t_m)\\
\pi_2 &= \texttt{Pattern}(\boxed{\texttt{==}}, \texttt{Pattern}(\boxed{\texttt{.}}, t_e, t_m), t_e)\\
s_1 &= \texttt{Absolute("1")} \qquad s_2 = \texttt{Absolute("2")}
\end{aligned}$$

and t_e and t_m are defined in Example 1. This operation updates the selected location x with a fresh call to `IsKind`, performed on the extracted receiver AST from x, and with the extracted right-hand side AST from x as its argument.

B. Synthesis algorithm

We now describe our algorithm for synthesizing AST transformations from input-output examples. Formally, it solves the following problem: given an example-based spec φ, find a transformation $T \in \mathcal{L}_{\text{T}}$ that is consistent with all examples $(P_i, P_o) \in \varphi$. We denote this problem as $T \vDash \varphi$. A transformation T is consistent with φ if and only if applying T to P_i produces the concrete edits that yield P_o from P_i.

Recall that the core methodology of PBE in PROSE is *deductive synthesis*, or *backpropagation*. In it, a problem of kind $F(T_1, T_2) \vDash \varphi$ is reduced to several subproblems of kinds $T_1 \vDash \varphi_1$ and $T_2 \vDash \varphi_2$, which are then solved recursively. Here φ_1 and φ_2 are fresh specs, which constitute necessary (or even sufficient) constraints on the subexpressions T_1 and T_2 in order for the entire expression $F(T_1, T_2)$ to satisfy φ. In other words, the examples on an operator F are backpropagated to examples on the parameters of F.

As discussed previously, the backpropagation algorithm relies on a number of modular operator-specific annotations called *witness functions*. Even though PROSE includes many generic operators with universal witness functions out of the box (e.g., list-processing `Filter`), most operators in $\mathcal{L}_{\text{T}}$ are domain-specific, and therefore require non-trivial domain-specific insight to enable backpropagation. The key part of this process is the witness function for the top-level `Transformation` operator.

The operator `Transformation`($rule_1$, …, $rule_n$) takes as input a list of rewrite rules and produces a transformation that, on a given AST, applies these rewrite rules in all applicable locations, producing a list of edits. The backpropagation problem for it is stated in reverse: given examples φ of edits performed on a given AST, find necessary constraints on the rewrite rules $rule_1$, …, $rule_n$ in the desired transformation.

The main challenges that lie in backpropagation for `Transformation` are:

1) Given an input-output example (P_i, P_o), which often represents the entire codebase/namespace/class, find examples of individual edits in the AST of P_i;
2) Partition the edits into clusters, deducing which of them were obtained by applying the same rewrite rule;
3) For each cluster, build a set of operation examples for the corresponding rewrite rule.

Finding individual edits We resolve challenge 1 by calculating *tree edit distance* between P_i and P_o. Note that the state-of-the-art Zhang-Shasha tree edit distance algorithm [24] manipulates single nodes, whereas our operations (and, consequently, examples of their behavior) manipulate whole subtrees. Thus, to construct proper examples for operations in $\mathcal{L}_{\text{T}}$, we group tree edits computed by the distance algorithm into connected components. A connected component of node edits represents a single edit operation over a subtree.

Partitioning into rewrite rules To identify subtree edits that were performed by the same rewrite rule, we use the DBSCAN [25] clustering algorithm to partition edits by similarity. Here we conjecture that components with similar edit distances constitute examples of the same rewrite rule.

Algorithm 1 describes the steps performed by the witness function for `Transformation`. Lines 2-6 perform the steps described above: computing tree edit distance and clustering the connected components of edits. Then, in lines 7-11, for each similar component, we extract the topmost operation to create an example for the corresponding rewrite rule. This example contains the subtree where the operation was applied in the input AST and the resulting subtree in the output AST.

C. Ranking

The last component of REFAZER is a ranking function for transformations synthesized by the backpropagation algorithm. Since $\mathcal{L}_{\text{T}}$ typically contains many thousands of ambiguous programs that are all consistent with a given example-based spec, we must disambiguate among them. Our ranking function

Algorithm 1 Backpropagation procedure for the DSL operator `Transformation`($rule_1$, ..., $rule_n$).

```
Require: Example-based spec $\varphi$
1: $result$ := a dictionary for storing examples for each input
2: for all ($P_i$,$P_o$) in $\varphi$ do
3:     $examples$ := empty list of refined examples for edits
4:     $operations$ := TreeEditDistance($P_i$, $P_o$)
5:     $components$ := ConnectedComponents($operations$)
6:     $connectedOpsByEdits$ := DBScan($components$)
7:     for all $connectedOps \in connectedOpsByEdits$ do
8:         $ruleExamples$ := Map($connectedOps$,
               $\lambda\, ops \rightarrow$ create a single concrete operation based on $ops$)
9:         $examples$ += $ruleExamples$
10:    end for
11:    $result[P_i]$ += $examples$
12: end for
13: return $result$
```

selects a transformation that is more likely to be robust on unseen ASTs – that is, avoid false positive and false negative matches. It is based on the following principles:

- Favor `Reference` over `ConstNode`: a transformation that reuses a node from the input AST is more likely to satisfy the intent than one that constructs a constant AST.
- Favor patterns with non-root paths, that is patterns that consider surrounding context of a location. A transformation that selects its locations based on surrounding context is less likely to generate false positives.
- Among patterns with non-empty context, favor the shorter ones. Even though context helps prevent underfitting (i.e., false positive matches), over-specializing to large contexts may lead to overfitting (i.e., false negative matches).

We describe the complete set of witness and ranking functions on Refazer's website.

IV. Evaluation

In this section, we present two empirical studies to evaluate Refazer . First, we present an empirical study on learning transformations for fixing student submissions to introductory Python programming assignments (Section IV-A). Then, we present an evaluation of Refazer on learning transformations to apply repetitive edits to open-source C# projects (Section IV-B). The experiments were performed on a PC with a Core i7 processor and 16GB of RAM, running Windows 10.

A. Fixing introductory programming assignments

In this study, we use Refazer to learn transformations that describe how students modify an incorrect program to obtain a correct one. We then measure how often the learned transformations can be used to fix other students' incorrect submissions. Transformations that apply across students are valuable because they can be used to generate hints to students on how to fix bugs in their code; alternatively, they can also help teaching assistants (TAs) with writing better manual feedback. We focus our evaluation on the transfer of transformations and leave the evaluation of hint generation to future work.

Our goal is to investigate both the overall effectiveness of our technique, and to what extent learned transformations in an education scenario are problem-specific, or general in nature. If most transformations are general purpose, instructors might be able to provide them manually, once. However, if most transformations are problem-specific, automated techniques such as Refazer will be especially valuable. Concretely, we address the following research questions:

RQ1 How often can transformations learned from student code edits be used to fix incorrect code of other students in the *same* programming assignment?

RQ2 How often can transformations learned from student code edits be used to fix incorrect code of other students who are solving a *different* programming assignment?

Benchmark We collected data from the introductory programming course CS61A at UC Berkeley. As many as 1,500 students enroll in this course per semester, which has led the instructors to adopt solutions common to MOOCs such as autograders. For each homework problem, the teachers provide a black-box test suite and the students use these tests to check the correctness of their programs. The system logs a submission whenever the student runs the provided tests for a homework assignment. This log thus provides a history of all submissions. Our benchmark comprises four assigned problems (see Table I). For each problem, students had to implement a single function in Python. We filtered the log data to focus on students who had at least one incorrect submission, which is required to learn a transformation from incorrect to correct state. We analyzed 21,781 incorrect submissions, from up to 720 students.

Experimental setup For each problem, each student in the data set submitted one or more incorrect submissions and, eventually, a correct one. We used the last incorrect submission and the correct one as input-output examples to synthesize a program transformation and used the synthesized transformation to attempt fixing other student submissions. By selecting a pair of incorrect and correct submissions, we learn a transformation that changes the state of the program from incorrect to correct, fixing existing faults in the code. Students may have applied additional edits, such as refactorings, though. The transformation thus may contain unnecessary rules to fix the code. By learning from the last incorrect submission, we increase the likelihood of learning a transformation that is focused on fixing the existing faults. We leave for future work the evaluation of learning larger transformations from earlier incorrect submissions to correct submissions, and how these transformations can help to fix larger conceptual faults in the code.

We used the teacher-provided test suites to check whether a program was fixed. Therefore, our technique relies on test cases for evaluating the correctness of fixed programs. While reliance on tests is a fundamental limitation, when fixes are reviewed in an interactive setting, our technique can be used to discover the need for more test cases for particular assignments.

For RQ1, we considered two scenarios: *Batch* and *Incremental*. In the *Batch* scenario, for each assignment, we synthesize transformations for all but one student in the data set and use them to fix the incorrect submissions of the remaining student, in a leave-one-out cross-validation – i.e., we attempt to fix the submission of a student using only transformations learned using submissions of other students. This scenario simulates the situation in which instructors have data from one or more previous semesters. In the *Incremental* scenario, we

TABLE I: Our benchmarks and incorrect student submissions.

Assignment		Students	Incorrect submissions
Product	product of the first n terms	549	3,218
Accumulate	fold-left of the first n terms	668	6,410
Repeated	function composition, depth n	720	9,924
G	$G(n) = \sum_{i=1}^{3} i \cdot G(n-i)$	379	2,229

sort our data set by submission time and try to fix a submission using only transformations learned from earlier timestamps. This scenario simulates the situation in which instructors lack previous data. Here the technique effectiveness increases over time. For RQ2, we use all transformations learned in *Batch* from one assignment to attempt to fix the submissions for another assignment. We selected four combinations of assignments for this experiment, and we evaluated the submissions of up to 400 students in each assignment.

In general, each synthesized rule in the transformation may apply to many locations in the code. In our experiments, we apply each synthesized transformation to at most 500 combinations of locations. If a transformation can be applied to further locations, we abort and proceed to the next transformation.

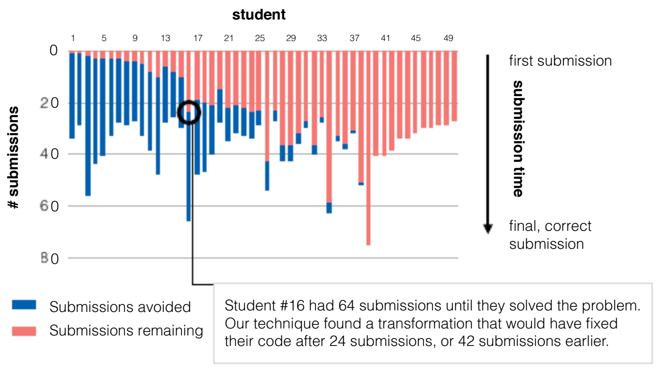
Learned transformations are useful within the same programming assignments In the *Batch* scenario, REFAZER generated fixes for 87% of the students. While, on average, students took 8.7 submissions to finish the assignment, the transformations learned using REFAZER fixed the student submissions after an average of 5.2 submissions. In the *Incremental* scenario, REFAZER generated fixes for 44% of the students and required, on average, 6.8 submissions to find a fix. The results suggest that the technique can be useful even in the absence of data from previous semesters but using existing data can double its effectiveness. Table II summarizes the results for both scenarios.

Although we only used students' last incorrect submissions and their corresponding correct submissions as examples for learning transformations, we could find a transformation to fix student solutions 3.5 submissions before the last incorrect submission, on average. This result suggests that REFAZER can be used to provide feedback to help students before they know how to arrive at a correct solution themselves. In addition, providing feedback about mistakes can be more important for students struggled with their assignments. Fig. 6 shows the 50 students with the highest number of submissions for the two hardest assignments. Each column shows chronological submissions for one student, with the earliest submissions at the top and the eventual correct submission at the bottom. Red indicates an incorrect submission; blue shows the first time REFAZER was able to fix the student's code (we only show the earliest time and do not re-test subsequent incorrect submissions). As we can see in the charts, students took dozens (up to 148) submissions. In many cases, REFAZER provided a fix after the student attempted half of the submissions.

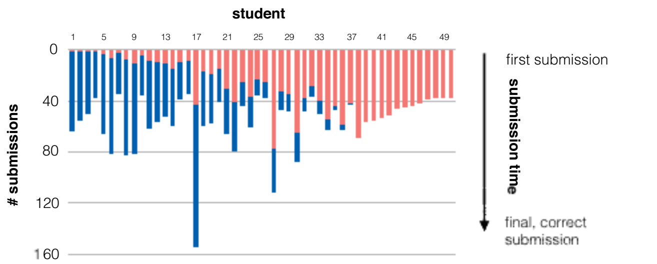
The transformations learned by REFAZER contain edits with different granularity, ranging from edits to single nodes in the AST, e.g., updating a constant, to edits that add multiple statements, such as adding a base case, a return statement, or even replacing an iterative solution by a recursive one. On average, the tree edit distance between the AST of an incorrect submission and the fixed one was 4.9 ($\sigma = 5.1$). However,

TABLE II: Summary of results for RQ1. "Incorrect submissions" = mean (SD) of submissions per student; "Students" = % of students with solution fixed by REFAZER; "Submissions" = mean (SD) of submissions required to find the fix.

Assignment	Incorrect submissions	Batch		Incremental	
		Students	Submissions	Students	Submissions
Product	5.3 (8.2)	501 (91%)	3.57 (6.1)	247 (45%)	4.1 (6.7)
Accumulate	8.9 (10.5)	608 (91%)	5.4 (7.9)	253 (38%)	7.5 (9.8)
Repeated	12.7 (15.3)	580 (81%)	8 (10.3)	340 (47%)	9.6 (11.5)
G	5.5 (9.4)	319 (84%)	1.4 (1.7)	174 (46%)	4.1 (7)
Total	8.7 (12)	2,008 (87%)	5.2 (8.1)	1,014 (44%)	6.8 (9.7)



(a) Assignment "Accumulate"



(b) Assignment "Repeated"

Fig. 6: Analysis of the first time REFAZER can fix a student submission for the 50 students with most attempts for two benchmark problems. Blue: submissions that might be avoided by showing feedback from a fix generated by REFAZER.

REFAZER did learn some larger fixes. The maximum tree edit distance was 45. We also noticed transformations containing multiple rules that represent multiple mistakes in the code.

Most learned transformations are not useful among different programming assignments Using transformations learned from other assignments we fixed submissions for 7-24% of the students, which suggests that most transformations are problem-specific and not common among different assignments, as shown in Table III. Column Original Assignment shows the assignments where REFAZER learned the transformations, and Target Assignment shows the assignments where the transformations were applied to. Accumulate was the assignment with most fixed submissions. The Accumulate function is a generalization of the Product function, used to learn the transformations, which may be the reason for the higher number

TABLE III: Summary of results for RQ2.

Original Assignment	Target Assignment	Helped students
Product	G	28 out of 379 (7%)
Product	Accumulate	94 out of 400 (24%)
Product	Repeated	43 out of 400 (11%)
Accumulate	G	33 out of 379 (9%)

of fixed faults. In general, the results suggest that different assignments exhibit different fault patterns; therefore, problem-specific training corpora are needed. This finding also suggests that other automatic grading tools that use a fixed or user-provided fault rubric (e.g., AutoGrader [9]) are not likely to work on arbitrary types of assignments.

Qualitative feedback from teaching assistants To validate the quality of the learned transformations, we built a user interface that allows one to explore, for each transformation, the incorrect submissions that can be fixed with it. We asked a TA of the CS61a course to analyze the fixes found using REFAZER. The TA confirmed that fixes were generally appropriate, but also reported some issues. First, a single syntactic transformation may represent multiple distinct mistakes. For instance, a transformation that changes a literal to `1` was related to a bug in the stopping condition of a while loop in one student's code; and also to a bug in the initial value of a multiplication which would always result in `0` in another student's code. In this case, the TA found it hard to provide a meaningful description of the fault beyond "replace 0 with 1". If fixes are used to generate feedback, TAs will need additional tools to merge or split clusters of student submissions. Finally, some fixed programs passed the tests but the TA noticed some faults remained due to missing tests.

B. Applying repetitive edits to open-source C# projects

In this study, we use REFAZER to learn transformations that describe simple edits that have to be applied to many locations of a C# codebase. We then measure how often the learned transformation is the intended one and whether it is correctly applied to all the required code locations. Concretely, we address the following question:

RQ3 Can REFAZER synthesize transformations with repetitive edits to large open-source projects?

Benchmark We manually inspected 404 commits from three open-source projects: Roslyn, Entity Framework, and NuGet. The projects' sizes range from 150,000 to 1,500,000 lines of code. Starting from the most recent commit, the first two authors inspected commit *diff* files—i.e., the code before and after the edits. If similar edits appeared three or more times, we classified the edit as repetitive. We identified 56 scenarios: 27 in Roslyn, 12 in Entity Framework, and 17 in NuGet.

The number of edited locations in each scenario ranges from 3 to 60 ($median = 5$). Each project contains at least one scenario with more than 19 edited locations. In 14 (25%) out of the 56 scenarios, there are edited locations in more than one file, which is harder to handle correctly for developers. Finally, in 39 (70%) out of the 56 scenarios, the edits are complex and context-dependent, meaning that a simple search/replace strategy is not enough to correctly apply the edits to all the necessary locations. We measured the size of the edits used as input-output examples by calculating the tree edit distance between the input AST and the output AST. On average, the distance was 13.0 ($\sigma = 13.6$). The maximum distance was 76.

Experimental setup We selected the examples for REFAZER as follows. First we randomly sort the edits described in the diff information. Next we incrementally add them as examples to REFAZER: after each example, we check whether the learned transformation correctly applies all subsequent edits. If the transformation misses an edit or incorrectly applies it according to the diff information, we take the first discrepancy in the edit list and provide it as the next example. If the transformation applies edits not presented in the diff, we manually inspect them to check whether the developer missed a location or the locations were incorrectly edited.

Results Table IV summarizes our results. REFAZER synthesized transformations for 55 out of 56 scenarios. In 40 (71%) scenarios, the synthesized transformations applied the same edits as developers, whereas, in 16 scenarios, the transformations applied more edits than developers. We manually inspected these scenarios, and conclude that 7 transformations were correct (i.e., developers missed some edits). We reported them to the developers of the respective projects. As of this writing, they confirmed 3 of these scenarios. In one of them, developers merged our pull request. In another one, they are not planning to spend time changing the code because the non-changed locations did not cause issues in the tests. In the last scenario, developers confirmed that other locations may need similar edits, but they did not inform us whether they will apply them.

In 9 scenarios (16%), additional edits were incorrect and revealed two limitations of the current DSL. First, some edits require further analysis to identify locations to apply them. For example, in scenario 18, developers replaced the full name of a type by its simple name. However, in some classes, this edit conflicted with another type name. As future work, we plan to extend our DSL to support type analysis and preconditions for each transformation. The second limitation relates to our tree pattern matching. Some examples produced templates that were too general. For example, if two nodes have different numbers of children, we can currently only match them based on their types. To support this kind of pattern, we plan to include additional predicates in our DSL such as `Contains`, which does not consider the entire list of children, but checks if any of the children match a specific pattern.

Our technique, on average, required 2.9 examples to synthesize all transformations in a diff. The number of required examples may vary based on example selection. Additionally, changes in the ranking functions preferring more general patterns over more restrictive ones can also influence the number of examples. We leave a further investigation of example ordering and of our ranking system to future work.

Threats to validity With respect to construct validity, our initial baseline is the diff information between commits. Some repetitive edits may have been performed across multiple commits, or developers may not have changed all possible code fragments. Therefore, some similar edits in each scenario may be unconsidered. To reduce this threat, we manually inspect REFAZER's additional edits. Concerning internal validity,

TABLE IV: Evaluation Summary. Scope = scope of the transformation; Ex. = examples; Dev. = locations modified by developers; REFAZER = locations modified by REFAZER. Outcomes: ✓ = it performed the same edits as the developers; ★ = it performed more edits than the developers (manually validated as correct); ✗ = it performed incorrect edits; "—" = it did not synthesize a transformation.

Id	Project	Scope	Ex.	Dev.	REFAZER	Outcome
1	EF	Single file	2	13	13	✓
2	EF	Single file	5	10	10	✓
3	EF	Multiple files	3	19	20	★
4	EF	Single file	3	4	4	✓
5	EF	Single file	3	3	3	✓
6	EF	Single file	2	3	3	✓
7	EF	Single file	2	4	10	★
8	EF	Multiple files	2	8	8	✓
9	EF	Single file	2	3	3	✓
10	EF	Single file	2	12	12	✓
11	EF	Multiple files	4	5	5	✓
12	EF	Single file	2	3	3	✓
13	NuGet	Single file	2	4	4	✓
14	NuGet	Multiple files	2	4	16	✗
15	NuGet	Single file	3	3	3	✓
16	NuGet	Multiple files	2	31	88	★
17	NuGet	Single file	3	3	3	✓
18	NuGet	Multiple files	4	8	14	✗
19	NuGet	Single file	4	14	43	✗
20	NuGet	Single file	2	4	4	✓
21	NuGet	Multiple files	5	5	13	✗
22	NuGet	Single file	3	3	3	✓
23	NuGet	Single file	3	5	5	✓
24	NuGet	Single file	2	3	3	✓
25	NuGet	Single file	3	4	4	✓
26	NuGet	Single file	2	9	32	✗
27	NuGet	Single file	3	4	4	✓
28	NuGet	Multiple files	3	4	10	★
29	NuGet	Multiple files	4	12	79	✗
30	NuGet	Single file	2	3	21	★
31	Roslyn	Multiple files	5	7	7	✓
32	Roslyn	Multiple files	3	17	17	✓
33	Roslyn	Single file	3	6	6	✓
34	Roslyn	Single file	2	9	9	✓
35	Roslyn	Multiple files	3	26	744	★
36	Roslyn	Single file	2	4	4	✓
37	Roslyn	Single file	4	4	4	✓
38	Roslyn	Single file	8	14	14	✓
39	Roslyn	Single file	2	60	—	—
40	Roslyn	Single file	3	8	8	✓
41	Roslyn	Multiple files	3	15	15	✓
42	Roslyn	Single file	2	7	7	✓
43	Roslyn	Single file	5	13	14	✗
44	Roslyn	Single file	2	12	12	✓
45	Roslyn	Single file	2	4	4	✓
46	Roslyn	Single file	2	5	5	✓
47	Roslyn	Single file	2	3	3	✓
48	Roslyn	Single file	4	11	11	✓
49	Roslyn	Single file	2	5	5	✓
50	Roslyn	Single file	2	3	5	★
51	Roslyn	Single file	2	5	5	✓
52	Roslyn	Single file	2	3	3	✓
53	Roslyn	Single file	4	6	6	✓
54	Roslyn	Multiple files	2	15	49	✗
55	Roslyn	Single file	3	4	4	✓
56	Roslyn	Single file	2	4	4	✓

example selection may affect the number of examples needed to perform the transformation. To avoid bias, we randomly selected the examples. Additionally, we performed this experiment three times and did not see significant changes in the required number of examples. Finally, our corpus of repetitive changes may not be representative for other kinds of software systems.

V. RELATED WORK

Example-based program transformations Meng et al. [1, 20, 26] propose Lase, a technique for performing repetitive edits using examples. Lase uses clone detection, isomorphic subgraph extraction, and dependency analysis to identify methods to apply the repetitive edits and match context for transformation concretization. Lase only learns abstractions from differences in examples, learning the most specific generalization. In contrast, REFAZER learns abstract transformations even from one example or non-identical examples. This is possible because REFAZER learns a set of transformations that satisfy the examples, not only the most specific one. Currently, both techniques are complementary with respect to applicability. Lase learns statement-level edits and cannot learn from repetitive edits that appear in different statement types (e.g., Fig. 1). Additionally, some repetitive edits in our benchmarks (e.g., Fig. 2) appear more than once in the same method, and Lase applies the learned transformation to at most one location per method.

Furthermore, in Lase's benchmark, the examples may diverge in terms of applied edits; that is, one example may have an update operation and a delete operation, and the other example may have just an update operation. Lase only uses common edits between examples, which means it can learn transformations from inconsistent examples but it does not learn example-specific edits. Currently, REFAZER cannot handle these examples, but we plan to extend the DSL operators to group examples according to their similarities and learn transformations for each group of examples separately.

Other approaches are semi-automatic using examples in combination with transformation templates [27, 28]. Unlike these techniques, our approach is fully automated. Feser et al. [29] propose a technique for synthesizing data structure transformations from examples in functional programming languages. Nguyen et al. [30] present LibSync, a technique that migrates APIs based on migrated clients. Tansey and Tilevich [31] present an example-based technique to migrate APIs that are based on annotations. HelpMeOut [32] learns transformations to fix compilation and run-time errors from examples. Asaduzzaman et al. [33] present Parc, a technique to recommend argument (parameter) for method calls based on parameter usage history and static type analyses. Unlike these techniques, REFAZER is not tailored to a specific domain.

Code completion techniques recommend code transformation while source code is edited. Raychev et al. [34] use data from large code repositories to learn likely code completions. Similarly, Foster et al. [35] use a large dataset of common code completions and recommend them based on the code context. Ge et al. [36] propose a similar technique for auto-completing a refactoring started manually. While these techniques are limited by IDE refactorings and their datasets, REFAZER can automate unseen transformations. Nguyen et al. [37] present APIRec, a technique that leverages repetitive edit properties to recommend API usage. APIRec learns a statistical model based on the co-occurrence of fine-grained changes and recommends API calls based on edit context. Slang [38] is a technique that computes a statistical model based on code fragments from

GitHub repositories. It receives as input a program with missing fragments to be inserted (a role) and relies on a synthesizer to fill in these fragments. While these techniques require a large codebase to learn statistical models, REFAZER can learn program transformations from one or few examples.

Inductive programming (IP) IP has been an active research area in the AI and HCI communities for over a decade [39]. IP techniques have recently been developed for various domains including interactive synthesis of parsers [40], imperative data structure manipulations [41], and network policies [42]. Recently, it has been successfully used in industry by FlashFill and FlashExtract [11, 12, 43]. FlashFill is a feature in Microsoft Excel 2013 that uses IP methods to automatically synthesize string transformation macros from input-output examples. FlashExtract is a tool for data extraction from semi-structured text files, deployed in Microsoft PowerShell for Windows 10 and as the Custom Field and Custom Log features in Operations Management Suite (a Microsoft log analytics tool). The REFAZER DSL is inspired by the DSLs of FlashExtract and FlashFill. While FlashFill uses the `ConstString` operator to create new strings and the `SubString` operator to get substrings from the input string, we use `ConstNode` and `Reference` operators to compose the new subtree using new nodes or nodes from the existing AST. In addition, our DSL contains specific operators for performing tree edits and tree pattern matching. FlashFill and FlashExtract motivated research on PROSE [14]. While PROSE has been primarily used in the data wrangling domain, our technique shows its applicability to a novel unrelated domain – learning program transformations.

Synthesis for education Singh et al. [9] propose AutoGrader, a program synthesis technique for fixing incorrect student submissions. Given a set of transformations that represent fixes for student mistakes (error model) and an incorrect submission, AutoGrader uses symbolic execution to try all combinations of transformations to fix the student submission. While AutoGrader requires an error model, REFAZER automatically generates it from examples of fixes. In the future, we plan to use the symbolic search of AutoGrader to efficiently explore all REFAZER transformations. Rivers and Koedinger [44] propose a data-driven technique for hint generation. The main idea is to generate concrete edits from the incorrect solution to the closest correct one. While they focus on comparing the entire AST, which can have many differences, our technique generalizes transformations that fix specific mistakes in student submissions. Kaleeswaran et al. [45] propose a semi-supervised technique for feedback generation. The technique clusters the solutions based on the strategies to solve it. Then instructors manually label one correct submission in each cluster. They formally validate incorrect solutions against correct one. Although our technique is completely automatic, we plan to investigate the use of formal verification to validate the transformations.

Program repair Automated program repair is the task of automatically changing incorrect programs to make them meet a desired specification [46]. One of the main challenges is to efficiently search program space to find one that behaves correctly. The most prominent search techniques are enumerative or data-driven. GenProg uses genetic programming to repeatedly alter the incorrect program aiming to correct it [47]. Data-driven approaches leverage large code repositories to synthesize likely changes to the input program [38]. Prophet [48] is a patch generation system that learns a probabilistic application-independent model of correct code from a set of successful human patches. Qlose provides ways to rank possible repairs based on a cost metric [49]. Unlike these techniques, which use a global model of possible transformations, REFAZER learns specific transformations using examples of code modification — i.e., from both the original and the modified program.

VI. CONCLUSIONS

We presented REFAZER, a technique for synthesizing syntactic transformations from examples. Given a set of examples consisting of program edits, REFAZER synthesizes a transformation that is consistent with the examples. Our synthesizer builds on the state-of-the-art program synthesis engine PROSE. To enable it, we develop (i) a novel DSL for representing program transformations, (ii) domain-specific constraints for the DSL operators, which reduce the space of search for transformations, and (iii) ranking functions for transformation robustness, based on the structure of the synthesized transformations. We evaluated REFAZER on two applications: synthesizing transformations that describe how students "fix" their programming assignments and synthesizing transformations that apply repetitive edits to large codebases. Our technique learned transformations that automatically fixed the program submissions of 87% of the students participating in a large UC Berkeley class and it learned the transformations necessary to apply the correct code edits for 84% of the repetitive tasks we extracted from three large code repositories.

As future work, we plan to increase the expressiveness of our tree pattern expressions to avoid selecting incorrect locations due to over-generalization. We aim at investigating the use of control-flow and data-flow analyses for identifying the context of the transformation, and the inclusion of negative examples and operators to specify undesired transformations. Although REFAZER is capable of fixing student bugs automatically, it lacks the domain knowledge of a teacher and can therefore generate functionally correct but stylistically bad fixes. To address this limitation, we have recently built a mixed-initiative approach that uses teacher expertise to better leverage the fixes produced by REFAZER [50]. Teachers can write feedback about an incorrect submission or a cluster of incorrect submissions. The system propagates the feedback to all other students who made the same mistake.

In addition to being a useful tool, REFAZER makes two novel achievements in PBE. First, it is the first application of backpropagation-based PBE methodology to a domain unrelated to data wrangling or string manipulation. Second, in its domain it takes a step towards development of fully unsupervised PBE, as it automates extraction of input-output examples from the datasets (that is, students' submissions or developers' modifications). We hope that with our future work on incorporating flow analyses into witness functions, REFAZER will become the first major application of inductive programming that leverages research developments from the entire field of software engineering.

ACKNOWLEDGMENT

This research was supported by the NSF Expeditions in Computing award CCF 1138996, NSF CAREER award IIS 1149799, CNPq/CAPES and a Google CS Capacity Award.

References

[1] N. Meng, M. Kim, and K. S. McKinley, "LASE: Locating and applying systematic edits by learning from examples," in *Proceedings of the 35th International Conference on Software Engineering*, ser. ICSE '13. Piscataway, NJ, USA: IEEE Press, 2013, pp. 502–511.

[2] L. Wasserman, "Scalable, example-based refactorings with refaster," in *Proceedings of the 2013 ACM Workshop on Workshop on Refactoring Tools*, ser. WRT '13. New York, NY, USA: ACM, 2013, pp. 25–28.

[3] Microsoft, "Visual Studio," 2016, at https://www.visualstudio.com.

[4] The Eclipse Foundation, "Eclipse," 2016, at https://eclipse.org/.

[5] JetBrains, "ReSharper," 2016, at https://www.jetbrains.com/resharper/.

[6] Synopsys, Inc., "Coverity," 2016, at http://www.coverity.com/.

[7] Google, "Error-prone," 2016, at http://errorprone.info/.

[8] Google, "Clang-tidy," 2016, at http://clang.llvm.org/extra/clang-tidy/.

[9] R. Singh, S. Gulwani, and A. Solar-Lezama, "Automated feedback generation for introductory programming assignments," in *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '13. New York, NY, USA: ACM, 2013, pp. 15–26.

[10] S. Gulwani, J. Hernández-Orallo, E. Kitzelmann, S. H. Muggleton, U. Schmid, and B. Zorn, "Inductive programming meets the real world," *Communications of the ACM*, vol. 58, no. 11, pp. 90–99, 2015.

[11] S. Gulwani, "Automating string processing in spreadsheets using input-output examples," in *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '11. New York, NY, USA: ACM, 2011, pp. 317–330.

[12] V. Le and S. Gulwani, "FlashExtract: A framework for data extraction by examples," in *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '14, New York, NY, USA, 2014, pp. 542–553.

[13] D. Edge, S. Gulwani, N. Milic-Frayling, M. Raza, R. Adhitya Saputra, C. Wang, and K. Yatani, "Mixed-initiative approaches to global editing in slideware," in *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems*, ser. CHI '15. New York, NY, USA: ACM, 2015, pp. 3503–3512.

[14] O. Polozov and S. Gulwani, "FlashMeta: A framework for inductive program synthesis," in *Proceedings of the ACM International Conference on Object-oriented Programming Systems, Languages, and Applications*, ser. OOPSLA '15. New York, NY, USA: ACM, 2015, pp. 542–553.

[15] A. Hindle, E. T. Barr, Z. Su, M. Gabel, and P. Devanbu, "On the naturalness of software," in *Proceedings of the 34th International Conference on Software Engineering*, ser. ICSE '12. Piscataway, NJ, USA: IEEE Press, 2012, pp. 837–847.

[16] G. Kniesel and H. Koch, "Static composition of refactorings," *Science of Computer Programming*, vol. 52, no. 1-3, pp. 9–51, 2004.

[17] Microsoft, "Project Roslyn," 2011, at https://github.com/dotnet/roslyn.

[18] Microsoft, "Entity Framework 6," at http://www.asp.net/entity-framework.

[19] Microsoft, "NuGet 2," at https://github.com/nuget/nuget2.

[20] N. Meng, M. Kim, and K. S. McKinley, "Systematic editing: Generating program transformations from an example," in *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '11. New York, NY, USA: ACM, 2011, pp. 329–342.

[21] R. Alur, R. Bodik, G. Juniwal, M. M. K. Martin, M. Raghothaman, S. A. Seshia, R. Singh, A. Solar-Lezama, E. Torlak, and A. Udupa, "Syntax-guided synthesis," in *Proceedings of the 2013 Formal Methods in Computer-Aided Design*, ser. FMCAD '13. Piscataway, NJ, USA: IEEE Press, 2013, pp. 1–8.

[22] World Wide Web Consortium, "XPath," 1999, at https://www.w3.org/TR/xpath/.

[23] M. Pawlik and N. Augsten, "RTED: A robust algorithm for the tree edit distance," *Proceedings of the VLDB Endowment*, vol. 5, no. 4, pp. 334–345, Dec. 2011.

[24] K. Zhang and D. Shasha, "Simple fast algorithms for the editing distance between trees and related problems," *SIAM Journal on Computing*, vol. 18, no. 6, pp. 1245–1262, 1989.

[25] M. Ester, H.-P. Kriegel, J. Sander, and X. Xu, "A density-based algorithm for discovering clusters in large spatial databases with noise," in *Proceedings of the Second International Conference on Knowledge Discovery and Data Mining*. AAAI Press, 1996, pp. 226–231.

[26] N. Meng, L. Hua, M. Kim, and K. S. McKinley, "Does automated refactoring obviate systematic editing?" in *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ser. ICSE '15. Piscataway, NJ, USA: IEEE Press, 2015, pp. 392–402.

[27] R. Robbes and M. Lanza, "Example-based program transformation," in *Model Driven Engineering Languages and Systems*, ser. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2008, vol. 5301, pp. 174–188.

[28] M. Boshernitsan, S. L. Graham, and M. A. Hearst, "Aligning development tools with the way programmers think about code changes," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, ser. CHI '07. New York, NY, USA: ACM, 2007, pp. 567–576.

[29] J. K. Feser, S. Chaudhuri, and I. Dillig, "Synthesizing data structure transformations from input-output examples," in *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '15. New York, NY, USA: ACM, 2015.

[30] H. A. Nguyen, T. T. Nguyen, G. Wilson, Jr., A. T. Nguyen, M. Kim, and T. N. Nguyen, "A graph-based approach to API usage adaptation," in *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, ser. OOPSLA '10. New York, NY, USA: ACM, 2010, pp. 302–321.

[31] W. Tansey and E. Tilevich, "Annotation refactoring: Inferring upgrade transformations for legacy applications," in *Proceedings of the 23rd ACM SIGPLAN Conference*

on Object-oriented Programming Systems Languages and Applications, ser. OOPSLA '08. New York, NY, USA: ACM, 2008, pp. 295–312.

[32] B. Hartmann, D. MacDougall, J. Brandt, and S. R. Klemmer, "What would other programmers do: Suggesting solutions to error messages," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, ser. CHI '10. New York, USA: ACM, 2010, pp. 1019–1028.

[33] M. Asaduzzaman, C. K. Roy, S. Monir, and K. A. Schneider, "Exploring API method parameter recommendations," in *Proceedings of the 31st IEEE International Conference on Software Maintenance and Evolution (ICSME)*, ser. ICSME '15. Washington, DC, USA: IEEE Computer Society, 2015, pp. 271–280.

[34] V. Raychev, M. Schäfer, M. Sridharan, and M. Vechev, "Refactoring with synthesis," in *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages &#38; Applications*, ser. OOPSLA '13. New York, NY, USA: ACM, 2013, pp. 339–354.

[35] S. R. Foster, W. G. Griswold, and S. Lerner, "WitchDoctor: IDE support for real-time auto-completion of refactorings," in *Proceedings of the 34th International Conference on Software Engineering*, ser. ICSE '12. Piscataway, NJ, USA: IEEE Press, 2012, pp. 222–232.

[36] X. Ge, Q. L. DuBose, and E. Murphy-Hill, "Reconciling manual and automatic refactoring," in *Proceedings of the 34th International Conference on Software Engineering*, ser. ICSE '12. Piscataway, NJ, USA: IEEE Press, 2012, pp. 211–221.

[37] A. T. Nguyen, M. Hilton, M. Codoban, H. A. Nguyen, L. Mast, E. Rademacher, T. N. Nguyen, and D. Dig, "API code recommendation using statistical learning from fine-grained changes," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2016. New York, NY, USA: ACM, 2016, pp. 511–522.

[38] V. Raychev, M. Vechev, and E. Yahav, "Code completion with statistical language models," in *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '14. New York, NY, USA: ACM, 2014, pp. 419–428.

[39] H. Lieberman, *Your wish is my command: Programming by example*. Morgan Kaufmann, 2001.

[40] A. Leung, J. Sarracino, and S. Lerner, "Interactive parser synthesis by example," in *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17*, 2015, pp. 565–574.

[41] J. K. Feser, S. Chaudhuri, and I. Dillig, "Synthesizing data structure transformations from input-output examples," in *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '15. New York, NY, USA: ACM, 2015.

[42] Y. Yuan, R. Alur, and B. T. Loo, "NetEgg: Programming network policies by examples," in *Proceedings of the 13th ACM Workshop on Hot Topics in Networks*, 2014, pp. 20:1–20:7.

[43] M. Mayer, G. Soares, M. Grechkin, V. Le, M. Marron, O. Polozov, R. Singh, B. Zorn, and S. Gulwani, "User interaction models for disambiguation in programming by example," in *Proceedings of the 28th Annual ACM Symposium on User Interface Software &#38; Technology*, ser. UIST '15. New York, NY, USA: ACM, 2015, pp. 291–301.

[44] K. Rivers and K. R. Koedinger, "Data-driven hint generation in vast solution spaces: a self-improving Python programming tutor," *International Journal of Artificial Intelligence in Education*, pp. 1–28, 2015.

[45] S. Kaleeswaran, A. Santhiar, A. Kanade, and S. Gulwani, "Semi-supervised verified feedback generation," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2016. New York, NY, USA: ACM, 2016, pp. 739–750.

[46] C. Goues, S. Forrest, and W. Weimer, "Current challenges in automatic software repair," *Software Quality Journal*, vol. 21, no. 3, pp. 421–443, 2013.

[47] C. Le Goues, M. Dewey-Vogt, S. Forrest, and W. Weimer, "A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each," in *Proceedings of the 34th International Conference on Software Engineering*, ser. ICSE '12. Piscataway, NJ, USA: IEEE Press, 2012, pp. 3–13.

[48] F. Long and M. Rinard, "Automatic patch generation by learning correct code," *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, vol. 51, no. 1, pp. 298–312, 2016.

[49] L. D'Antoni, R. Samanta, and R. Singh, *Qlose: Program repair with quantitative objectives*. Springer International Publishing, 2016, pp. 383–401.

[50] A. Head, E. Glassman, G. Soares, R. Suzuki, L. D'Antoni, and B. Hartmann, "Writing Reusable Code Feedback at Scale with Mixed-Initiative Program Synthesis," in *L@S'17: 4th ACM Conference on Learning at Scale*, 2017.