Feedback-Driven Semi-Supervised Synthesis of Program Transformations

XIANG GAO*, National University of Singapore, Singapore
SHRADDHA BARKE*, University of California, San Diego, USA
ARJUN RADHAKRISHNA, Microsoft, USA
GUSTAVO SOARES, Microsoft, USA
SUMIT GULWANI, Microsoft, USA
ALAN LEUNG, Microsoft, USA
NACHIAPPAN NAGAPPAN, Microsoft Research, USA
ASHISH TIWARI, Microsoft, USA

While editing code, it is common for developers to make multiple related repeated edits that are all instances of a more general program transformation. Since this process can be tedious and error-prone, we study the problem of automatically learning program transformations from past edits, which can then be used to predict future edits. We take a novel view of the problem as a semi-supervised learning problem: apart from the concrete edits that are instances of the general transformation, the learning procedure also exploits access to additional inputs (program subtrees) that are marked as positive or negative depending on whether the transformation applies on those inputs. We present a procedure to solve the semi-supervised transformation learning problem using anti-unification and programming-by-example synthesis technology. To eliminate reliance on access to marked additional inputs, we generalize the semi-supervised learning procedure to a feedback-driven procedure that also generates the marked additional inputs in an iterative loop. We apply these ideas to build and evaluate three applications that use different mechanisms for generating feedback. Compared to existing tools that learn program transformations from edits, our feedback-driven semi-supervised approach is vastly more effective in successfully predicting edits with significantly lesser amounts of past edit data.

CCS Concepts: • **Software and its engineering → Automatic programming**; **Software maintenance tools**; **Integrated and visual development environments**; • **Computing methodologies → Artificial intelligence**.

Additional Key Words and Phrases: Program transformation, Refactoring, Program synthesis, Programming by Example

ACM Reference Format:
Xiang Gao, Shraddha Barke, Arjun Radhakrishna, Gustavo Soares, Sumit Gulwani, Alan Leung, Nachiappan Nagappan, and Ashish Tiwari. 2020. Feedback-Driven Semi-Supervised Synthesis of Program Transformations. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 219 (November 2020), 30 pages. https://doi.org/10.1145/3428287

*Xiang Gao and Shraddha Barke worked on this paper during their internships with the Prose team at Microsoft.

Authors' addresses: Xiang Gao, National University of Singapore, Singapore, gaoxiang@comp.nus.edu.sg; Shraddha Barke, University of California, San Diego, USA, sbarke@eng.ucsd.edu; Arjun Radhakrishna, Microsoft, USA, arradha@microsoft.com; Gustavo Soares, Microsoft, USA, gustavo.soares@microsoft.com; Sumit Gulwani, Microsoft, USA, sumitg@microsoft.com; Alan Leung, Microsoft, USA, alan.leung@microsoft.com; Nachiappan Nagappan, Microsoft Research, USA, nachin@microsoft.com; Ashish Tiwari, Microsoft, USA, ashish.tiwari@microsoft.com.

2475-1421/2020/11-ART219
https://doi.org/10.1145/3428287

1 INTRODUCTION

Integrated Development Environments (IDEs) and static analysis tools help developers edit their code by automating common classes of edits, such as boilerplate code edits (e.g., equality comparisons or constructors), code refactorings (e.g., rename class, extract method), and quick fixes (e.g., fix possible `NullReferenceException`). To automate these edits, tool builders implement *code transformations* that manipulate the Abstract Syntax Tree (AST) of the user's code to produce the desired code edit.

While traditional tools support a predefined catalog of transformations handcrafted by tool builders, in recent years, we have seen an emerging trend of tools and techniques that synthesize program transformations using examples of code edits [Bader et al. 2019; Meng et al. 2011, 2013; Miltner et al. 2019; Rolim et al. 2017, 2018]. For instance, Getafix [Bader et al. 2019] learns fixes for static analysis warnings using previous fixes as examples. It has been deployed at Facebook where it is used for the maintenance of Facebook apps. BluePencil [Miltner et al. 2019] produces code edit suggestions to automate repetitive code edits, i.e., edits that follow the same structural pattern but that may involve different expressions. It synthesizes transformations on-the-fly based on the recent edits performed by the developer. BluePencil has been released in Microsoft Visual Studio 2019 [Microsoft 2019] and is available as Visual Studio IntelliCode suggestions [Microsoft 2020].

The main challenge of generalizing examples of edits to program transformations lies in synthesizing an intended generalization that not only satisfies the given examples but also produces the correct edits on unseen inputs. Incorrect generalizations can lead to *false negatives*: the transformation does not produce an edit suggestion in a location that should be changed. False negatives increase the burden on developers, since it requires developers to either provide more examples or perform the edits themselves, reducing the number of automated edits. Moreover, it may cause developers to miss edits leading to bugs and inconsistencies in the code. Incorrect generalizations can also lead to *false positives*: the transformation produces an incorrect edit. While false negatives are usually related to transformations that are too specific, false positives are mostly related to transformations that are too general. Both false negatives and positives can reduce developers' confidence in the aforementioned systems, and thus, finding the correct generalization is crucial for the adoption of these systems.

Existing approaches have tried to handle the generalization problem in different ways. Sydit [Meng et al. 2011] and Lase [Meng et al. 2013] can only generalize names of variables, methods and fields when learning a code transformation. The former only accepts one example and synthesizes the transformation using the most general generalization. The latter accepts multiple examples and synthesizes the transformation using the most specific generalization, which is also the approach adopted by Revisar [Rolim et al. 2018] and Getafix [Bader et al. 2019]. Using either the most specific or the most general generalization is usually undesirable, as they are likely to produce false negatives and false positives, respectively. ReFazer [Rolim et al. 2017] learns a set of transformations consistent with the examples and stores them as a Version Space Algebra (VSA) [Mitchell 1982]. It then uses a ranking system to rank the transformations and selects the one that is more likely to be correct based on a set of predefined heuristics. However, despite the more sophisticated approach to generalization, in certain cases, ReFazer still requires up to six examples of a repetitive edit before producing edit suggestions [Rolim et al. 2017].

All aforementioned techniques rely only on input-output examples of edits and background knowledge in the form of ranking schemes and heuristics to deal with the generalization problem. However, apart from these, an additional source of information could be the large number of *additional input trees* available in the remainder of the file and project the user is editing. Semi-supervised learning [Zhu and Goldberg 2009] is an approach to machine learning that combines a set of labeled input-output examples and unlabeled data (inputs) during training. It has recently become more popular and practical due to the variety of problems for which vast quantities of unlabeled data are available, e.g. text on websites, protein sequences, or images [Zhu 2005]. The fact that many additional inputs are available in source code inspires a natural question:

Is it possible to combine input-output examples with additional inputs to synthesize program transformations?

Our first key observation is that an additional input AST can help us disambiguate how to generalize the transformation by providing more examples of ASTs that should be manipulated by the transformation. Consider a simple change from `if (score < limit)` to `if (IsValid(score))`. With a single example, it is not clear whether we do the transformation only when the left-hand side of the comparison is `score`. However, if one says that the transformation should also apply to `if (GetScore(run) < limit)`, then we have one more example for the LHS expression, `GetScore(run)`, and we can use this example to refine our transformation—in this case, generalize it further. However, we still need to identify the locations in the source code (the additional inputs) where the transformation should apply. Our second key observation is that we can predict whether an arbitrary input should be an additional input by evaluating the quality of the transformation synthesized when using the new input. The quality is assessed using a user-driven or automated feedback system.

We propose a feedback-driven semi-supervised technique to synthesize program transformations. The proposed approach is based on our two key observations above. Initially, our technique synthesizes a program transformation from input-output examples using ReFazer [Rolim et al. 2017]. For the input-output example, it tracks which subtrees of the AST (corresponding to a sub-expression) were used to construct the output, and can potentially be generalized. We call these nodes *selected nodes*. As an example, consider again the change `if (score < limit)` to `if (IsValid(score))`. The expression `score` was used in the output—it is a selected node. Next, our technique iterates over candidate additional inputs to find more examples to refine the generalization. For each candidate input, it performs two main steps:

- First, our technique computes the *anti-unification* of the examples and the candidate additional input. Anti-unification is a generalization process which can identify corresponding subtrees among different input ASTs. For instance, it can identify that `score` in the example input corresponds to `GetScore(run)` in the candidate additional input `if (GetScore(run) < limit)`. Our anti-unification based generalization algorithm tries to compute a generalization where each selected node in the example input has a corresponding node in the candidate additional input. For example, if the candidate additional input was `if (UnrelatedCondition())`, then we can infer the correspondence between `(score < limit)` and `(UnrelatedCondition())`, and the subtree `score` itself has no corresponding subtree, which causes anti-unification to fail to find a generalization. If anti-unification fails, the candidate additional input is not compatible and we discard it. Otherwise, we generate a new example from the candidate additional input, and re-synthesize parts of the transformation while taking this example into consideration. In our running scenario, the new example is `if (GetScore(run) < limit)` ↦ `if (IsValid(GetScore(run)))`.
- Then, our technique uses a feedback system to further evaluate whether the current candidate input should be accepted. The feedback is provided by a *reward function* that can be composed of different components. It can take into consideration user-provided feedback, for example, if the transformation should apply to a particular input. Indicating such inputs is usually an easier task for the user than providing another input-output example. However, the feedback can also use automated components based on, for example, the similarity of the additional input to the example inputs. If the final reward score is above a certain threshold, it accepts the additional input and synthesizes a new program transformation using the new example.

We implemented our technique for the domain of C# program transformations. It uses the implementation of ReFazer available in the PROSE SDK[1]. Further, we augmented the BluePencil algorithm [Miltner et al. 2019] with our approach to synthesize on-the-fly transformations. BluePencil provides a *modeless interface* where developers do not need to enter a special mode to provide examples, but instead, they are inferred from the history of changes to a particular file.

With these components, we implemented three applications that use feedback-driven semi-supervised synthesis:

- ReFazer*: *User-provided feedback about additional inputs.* This application allows developers to specify, as an additional input, a subtree where the transformation did not produce an edit (false negative). This implementation is motivated by the fact that when the transformation-learning system produces a false negative, it is easier for the developer to provide an additional input rather than a complete input-output example. On a benchmark of 12,642 test cases, we compared ReFazer* with the baseline (ReFazer). While the recall of ReFazer ranged from 26.71% (1 example provided) to 89.10% (3 examples provided), the recall of ReFazer* was at least 99.94% and its precision was at least 96.01% with just 1 example and 1 additional input provided. These results suggest that ReFazer* can synthesize suggestions with high precision at locations indicated by developers as false negatives.
- BluePencil$_{\text{cur}}$: *Semi-automated feedback based on cursor position.* This feature uses the cursor position in the editor to indicate candidate additional inputs to semi-supervised synthesis. This feature is motivated by the fact that the developers may either not be aware that they can provide additional inputs (discoverability problem [Miltner et al. 2019]), or may not want to break their workflow to provide additional inputs. The cursor position acts as a proxy for the user and indicates, implicitly, that the user wants to modify the current location. However, the cursor location is ambiguous. The subtree that the user wants to edit may be any of the subtrees that are present at the cursor location, i.e., the lowest leaf node at the cursor location all the way to the root of the AST. The tool relies on feedback from a reward function (Section 4.2) to accept additional inputs. We compared this reward function with two alternative reward functions: (i) *no validation*, where semi-supervised synthesis accepts any additional inputs; (ii) and *clone detection* where semi-supervised synthesis accepts inputs based on their similarities with the inputs in the input-output examples. Our results show that while "no validation" and "clone detection" lead to high false positives and negatives, respectively, our reward function produces only 11 false positives and 14 false negatives on 243,682 tested additional inputs. We also evaluated the effectiveness of BluePencil$_{\text{cur}}$ in generating correct suggestions at the cursor location. Amongst 291 scenarios, BluePencil$_{\text{cur}}$ only generates one false positive and three false negatives.
- BluePencil$_{\text{auto}}$: *Fully-automated feedback based on all inputs in the source code.* This feature uses all the nodes available in the source code as input to semi-supervised synthesis. It is relevant in the settings where user feedback is not available. For example, (a) when the developer themselves may not be aware of all locations that must be changed, or (b) when the developer may want to apply the edits in bulk, instead of inspecting each one for correctness. We evaluated how often BluePencil$_{\text{auto}}$ can save developers from indicating the additional inputs. To do so, we simulated a developer performing 350 repetitive edits with BluePencil$_{\text{cur}}$ and BluePencil$_{\text{auto}}$ enabled or just BluePencil enabled. In our experiment, BluePencil$_{\text{auto}}$ decreased the number of times the developer would have to indicate the input by 30%. When compared to BluePencil, our results show that BluePencil$_{\text{cur}}$ and BluePencil$_{\text{auto}}$ automated 263 edits while BluePencil automated only 159.

Contributions. We summarize the contributions of this paper as follows:

[1] https://www.microsoft.com/en-us/research/group/prose/

- We formalize the feedback-driven semi-supervised synthesis problem (Section 3);
- We propose semi-supervised synthesis for program transformations (Section 4), which is the first known semi-supervised synthesis technique in this field;
- We propose three practical applications based on semi-supervised synthesis and instantiate them for the domain of C# program transformations (Section 5);
- We evaluate our technique along the dual axes of effectiveness (quality as measured by false positive and negative rates) and efficiency (user burden as measured by the number of examples and additional inputs). Our results show that our technique achieves precision of over 96% with near-perfect recall across 86 real-world developer scenarios, all while delivering each suggestion in less than half a second.

Remark 1.1. In this paper, the term *semi-supervised* is used in a subtly different manner than in the traditional machine learning context. In both settings, additional unlabelled inputs are used to aid learning. However, in machine learning, the additional unlabelled inputs are used to understand the structure and distribution of the input space. On the other hand, in our setting, additional inputs are used to generate new input-output examples along the lines of existing labeled examples, using the structure of individual additional input trees. In other words, semi-supervised machine learning exploits the structure of the input space, while we use the structure of individual inputs.

2 MOTIVATING EXAMPLE

We start by illustrating the challenges of synthesizing code transformations from input-output examples. Consider the scenario shown in Figure 1.

A C# developer working on the NuGet[2] codebase has refactored the `ResolveDependency` method to make it static, then moved it to the new static class `DependencyResolveUtility`. As a result, the developer must update all invocations of this method to match its new signature. Figure 1a shows two call sites where the developer has manually updated the invocation to match this signature. Figures 1b and 1c show additional locations that will require a similar modification: note that they share the same general structure but contain dissimilar subexpressions. Manually performing such repetitive edits is tedious, error-prone, and time-consuming. Unfortunately, developer tools such as the Visual Studio IDE [Microsoft 2019] and ReSharper [JetBrains 2020b] do not include built-in transformations or refactorings to automate these edits.

However, a recently introduced Visual Studio feature based on BluePencil [Miltner et al. 2019], called IntelliCode suggestions (IntelliCode for brevity in the remainder of this paper), can learn to automate these edits after watching the developer perform a handful of edits. Specifically, after watching edits to the two locations shown in Figure 1a, IntelliCode learns a transformation and suggests automated edits to the locations shown in Figure 1b.

With only these two examples, however, IntelliCode is not yet able to produce suggestions for the locations shown in Figure 1c. These are *false negatives*. This is because the inputs in the examples provided so far differed only in their first method argument: `dependency1` and `dependency2`, respectively. As a result, IntelliCode synthesizes a transformation that generalizes across variation in the first argument, but not the others. While sufficient to suggest edits for the locations in Figure 1b, this transformation is not sufficiently general to apply to the locations shown in Figure 1c, which contain additional variation in the call target, third argument, and fifth argument (`Marker`, `AllowPrereleaseVersions`, and `Highest`, respectively).

To address this situation, the developer performs another manual edit at the first location in Figure 1c. IntelliCode consumes this edit as a new example and synthesizes a new transformation to generalize across variation in both the first and fifth arguments: IntelliCode has disambiguated

[2] Nuget is a package manager for .NET

(a) Two repetitive edits. Both edits update invocations to the method ResolveDependency but one of the arguments is different. Given these two edits, IntelliCode synthesizes a transformation to automate similar edits.

```
-  repository.ResolveDependency(dependency1, null, false, false, Lowest);
+  DependencyResolverUtility.ResolveDependency(repository, dependency1, null, false, false,
     Lowest);

-  repository.ResolveDependency(dependency2, null, false, false, Lowest);
+  DependencyResolverUtility.ResolveDependency(repository, dependency2, null, false, false,
     Lowest);
```

(b) IntelliCode correctly produces suggestions at these locations based on the previous edits. The first argument is the only difference between these locations, similar to the examples.

```
    repository.ResolveDependency(dependency3, null, false, false, Lowest);
    repository.ResolveDependency(dependency4, null, false, false, Lowest);
```

(c) IntelliCode fails to produce suggestions to these locations (false negative). Note that there are more elements that are different in these locations compared to the locations in the examples.

```
    repository.ResolveDependency(dependency1, null, false, false, Highest);
    repository.ResolveDependency(dependency2, null, false, false, Highest);
    Marker.ResolveDependency(dependency, null, AllowPrereleaseVersions, false,
        Highest);
```

(d) While this location shares the same structure as the previous ones, the transformation should not produce an edit here.

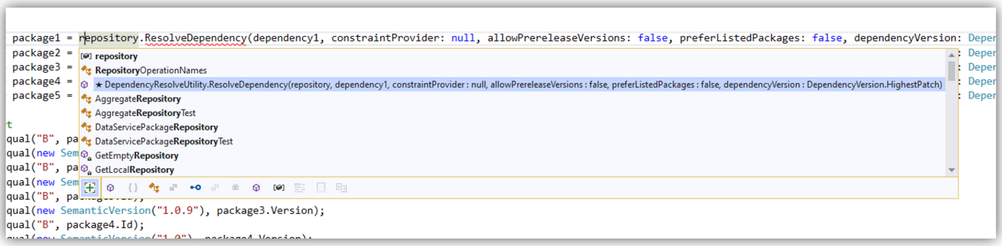
```
- s.GetUpdates(IsAny<IEnumerable<IPackage>>(), false, false,
+ DependencyResolverUtility.GetUpdates(s, IsAny<IEnumerable<IPackage>>(), false, false,
      IsAny<IEnumerable<FrameworkName>>(), IsAny<IEnumerable<IVersionSpec>>())
```

Fig. 1. A scenario with two repetitive edits (input-output examples), additional inputs, and a false positive. All inputs share the same structure (a method invocation with 5 arguments).

the developer's intent because the new example contains a different variable (Highest rather than Lowest) in the final argument. At this point, IntelliCode is now able to produce correct suggestions for all locations that differ only in their first or last argument. Unfortunately, despite having seen three input-output examples, it still fails to produce suggestions for the last location in Figure 1c.

In general, false negatives like those described stem from insufficiently general transformations–they overfit to the given examples. They not only reduce the applicability of the tool but also frustrate developers, who naturally expect an edit suggestion to automate their task after having already supplied several examples. The line between *too specific* and *too general* can be thin, though. In this scenario, the desired transformation should produce edits on invocations of the instance method ResolveDependency using 5 arguments. If we generalize the name of the method to any method, it will lead to false positives. For instance, it would produce the edit shown in Figure 1d.

Our Solution. We now illustrate how a system based on semi-supervised synthesis can help alleviate this problem. BLUEPENCIL$_{\text{cur}}$ uses the cursor position in the editor to indicate candidate additional inputs to our semi-supervised synthesis technique. Consider the first false negative shown in Figure 1c. As soon as the developer places the cursor in the location related to the false negative, BLUEPENCIL$_{\text{cur}}$ uses our semi-supervised feedback synthesis technique to improve the



```
package1 = repository.ResolveDependency(dependency1, constraintProvider: null, allowPrereleaseVersions: false, preferListedPackages: false, dependencyVersion: Deper
package2 =                                                                                                                                              : Deper
package3 =                                                                                                                                              : Deper
package4 =                                                                                                                                              : Deper
package5 =                                                                                                                                              : Deper

t
qual("B", pa
qual(new Sem
qual("B", pa
qual(new Sem
qual("B", pa
qual(new SemanticVersion("1.0.9"), package3.Version);
qual("B", package4.Id);
```

repository
RepositoryOperationNames
★ DependencyResolveUtility.ResolveDependency(repository, dependency1, constraintProvider : null, allowPrereleaseVersions : false, preferListedPackages : false, dependencyVersion : DependencyVersion.HighestPatch)
AggregateRepository
AggregateRepositoryTest
DataServicePackageRepository
DataServicePackageRepositoryTest
GetEmptyRepository
GetLocalRepository

Fig. 2. BLUEPENCIL$_{\text{cur}}$ implemented as a Visual Studio Extension. The developer clicks on a line to manually edit the code where the PBE system produced a false negative. BLUEPENCIL$_{\text{cur}}$ uses feedback-driven program-synthesis to synthesize a transformation that is general enough to be applied to this location. The edit generated by the transformation is shown as an auto-completion suggestion.

transformation. The new transformation produces an *auto-completion* suggestion for the current location (see Figure 2). We provide details of our technique and its applications in Sections 4 and 5, resp. In the next section, we formalize the problem of feedback-driven semi-supervised synthesis.

3 THE SEMI-SUPERVISED SYNTHESIS PROBLEM

We first formalize the semi-supervised synthesis problem and then discuss the feedback-driven semi-supervised synthesis problem.

3.1 Preliminaries and Problem Statements

Abstract Syntax Trees. Let $\mathbb{T}$ denote the set of all abstract syntax trees (AST). We use the notation t to denote a single AST in $\mathbb{T}$, and use the notation $\mathsf{SubTrees}(t) \subseteq \mathbb{T}$ to denote the set of all subtrees in t. Each node in the AST consists of a string label representing the node type (e.g., Identifier, MethodDeclaration, InvokeExpression, etc), set of attributes (e.g., text value of leaf nodes, etc) and a list of children ASTs.

Edit Programs. An *edit program*[3] $\mathsf{P} : \mathbb{T} \nrightarrow \mathbb{T}$ is a partial function[4] that maps ASTs to ASTs. In this paper, we assume that each edit program P is a pair $(\mathsf{P}_{\mathsf{guard}}, \mathsf{P}_{\mathsf{trans}})$ of two parts: (a) a *guard* $\mathsf{P}_{\mathsf{guard}} : \mathbb{T} \rightarrow \mathbb{B}$, and (b) a *transformer* $\mathsf{P}_{\mathsf{trans}} : \mathbb{T} \nrightarrow \mathbb{T}$. We have that $\mathsf{P}(t) = \mathsf{P}_{\mathsf{trans}}(t)$ when $\mathsf{P}_{\mathsf{guard}}(t)$ is true, and $\mathsf{P}(t) = \bot$ otherwise.

Example 3.1. Consider the two edits shown in Figure 1a. For each edit, the following edit program maps the subtree before the change to the subtree after the change.

$$\begin{aligned}
\mathsf{P}_{\mathsf{guard}} = {} & \textit{Input matches } \mathsf{X}_1.\mathsf{X}_2(\mathsf{X}_3, \mathsf{X}_4, \mathsf{X}_5, \mathsf{X}_6, \mathsf{X}_7) \ \textit{where} \\
& \quad \mid \mathsf{X}_1.\mathsf{label} = \mathsf{Identifier} \wedge \mathsf{X}_1.\mathsf{Attributes.TextValue} = \texttt{repository} \\
& \quad \mid \mathsf{X}_2.\mathsf{label} = \mathsf{Identifier} \wedge \mathsf{X}_2.\mathsf{Attributes.TextValue} = \texttt{ResolveDependency} \\
& \quad \mid \mathsf{X}_3.\mathsf{label} = \mathsf{X}_4.\mathsf{label} = \cdots = \mathsf{Argument} \wedge \mathsf{X}_4.\mathsf{Attributes}[\mathsf{TextValue}] = \texttt{null} \wedge \ldots \\
\mathsf{P}_{\mathsf{trans}} = {} & \textit{return } \texttt{DependencyResolveUtility}.\mathsf{X}_2(\mathsf{X}_1, \mathsf{X}_3, \mathsf{X}_4, \mathsf{X}_5, \mathsf{X}_6, \mathsf{X}_7)
\end{aligned}$$

[3] We also refer to edit programs more generally as *transformations*.
[4] In this paper, we consistently use $\nrightarrow$ to denote partial functions.

ReFazer learns this program initially in Section 2 (with just 2 examples). This program is written in terms of templates with each $\mathsf{X_i}$ representing a hole. In Section 3.2, we present a domain-specific language to express such programs.

The Semi-Supervised Synthesis Problem. As explained in Section 2, the semi-supervised synthesis problem is the core piece among the techniques in this paper. Semi-supervised synthesis allows a user or an environment to finely control the level of generalization used by the synthesizer. The formal definition of the problem is as follows. Given (a) a set of input-output examples $\mathsf{Examples} = \{\mathsf{i}_0 \mapsto \mathsf{o}_0, \ldots, \mathsf{i}_k \mapsto \mathsf{o}_k\}$, (b) a set of additional positive inputs $\mathsf{PI} = \{\mathsf{pi}_0, \ldots, \mathsf{pi}_n\}$, and (c) a set of additional negative inputs $\mathsf{NI} = \{\mathsf{ni}_0, \ldots, \mathsf{ni}_m\}$, the *semi-supervised synthesis problem* is to produce a program P such that (a) $\forall 0 \leq j \leq k. \mathsf{P}(\mathsf{i}_j) = \mathsf{o}_j$, (b) $\forall 0 \leq j \leq n. \mathsf{P}(\mathsf{pi}_j) \neq \bot$, and (c) $\forall 0 \leq j \leq m. \mathsf{P}(\mathsf{ni}_j) = \bot$. Intuitively, the problem asks for a program that is consistent with the provided examples, produces outputs on all additional positive inputs, and does not produce an output on any additional negative inputs. The over-generalization and under-generalization problem can be addressed by providing more additional negative and positive examples, respectively.

The Feedback-Driven Semi-Supervised Synthesis Problem. The semi-supervised synthesis problem assumes access to positive and negative additional inputs, but how do we find (more of) them to help refine the synthesized program? We use feedback from either the user or the environment to discover these additional inputs. In this setting, the synthesizer is provided with the following components: (a) A finite *pool of inputs* $\mathsf{InputPool} \subseteq \mathbb{T}$. We assume that all example inputs and additional (positive or negative) inputs are drawn from the input pool InputPool. In practice, the input pool is usually the set of all subtrees of the AST representing a source file. (b) A *reward function* $\mathsf{Rew} : \mathsf{InputPool} \not\to [-\infty, \infty]$ that acts as a feedback mechanism. A high and a low reward for an $\mathsf{i} \in \mathsf{InputPool}$ indicates whether the synthesized program should be applicable to i or not, respectively. For exposition purposes, we separate the reward function into the *user provided* $\mathsf{Rew_U}$ and *environment provided* $\mathsf{Rew_E}$ reward functions with Rew being a combination of the two. In Section 4.2, we define *feedback oracles* which take as input the state of the feedback loop (i.e., examples, positive and negative inputs, synthesized program) and return a reward function. While we could merge the notion of feedback oracle and reward function, with reward function taking additional inputs mentioned, this separation allows for easier notation.

The rewards are generated from a number of factors including (a) if the user manually indicates whether an input from the input pool should be positively or negatively marked, (b) whether applying a produced edit leaves the source code document in a compilable state, and (c) whether the produced edit for an input is similar to or different from the given examples.

This workflow proceeds in multiple rounds of interaction. In the n^{th} iteration of the workflow,

- The synthesizer, using the examples and the reward function Rew_{n-1}, produces a program P_n that is consistent with the examples Examples and the positive (and negative) additional inputs deduced from Rew_{n-1}.
- Optionally, the user adds new examples to the set of Examples to produce $\mathsf{Examples}_n$.
- The user and the environment in conjunction produce the rewards $\mathsf{Rew}_n : \mathsf{SubTrees}(t_n) \not\to [-\infty, \infty]$ to provide feedback on how P_n is to be refined in the next iteration to produce P_{n+1}.

This workflow is a continuous interaction between the environment and the user on one side, and the synthesizer on the other. This continuous interaction using rewards is reminiscent of a reinforcement learning scenario. However, in our setting, the user and the environment cannot be modeled as a Markov decision process, and the state space is non-continuous infinite, making standard reinforcement learning techniques not applicable.

Due to the user-in-the-loop nature of the feedback-driven semi-supervised synthesis workflow, it is infeasible to define an explicit correctness condition for the problem. The real optimality criterion

```
program := (guard, transformer)                 transformer := construct | select
guard   := pred | Conjunction(pred, guard)      construct      := Tree(kind, attrs, children)
pred    := IsNthChild(node, n)                  children       := EmptyChildren | Cons(node, children)
         | IsKind(node, kind)                      | InsertChild(Children(select), pos, node)
         | Attribute(node, attr) = value           | DeleteChild(Children(select), pos)
         | Not(pred)                               | ReplaceChildren(Children(select), posList, children)
                                                   | MapChildren(λ input: transformer, Children(select))
node    := Path(input, path)                    select         := Nth(Filter(guard, SubTrees(input)), n)
                                                pos            := n | ChildIndexOf(node)

Variables:
    AST input;       List<int> posList;          XPath path;
    int n;           string kind, attr, value;   Dictionary<string, string> attrs;
```

Fig. 3. Domain-specific language for edit programs

for the synthesized program is *how well does the synthesized program match user intent?* This criterion is hard to capture formally in any meaningful way. Further, depending on the scenario, the same program may either be correct or incorrect. For example, in the case from Section 2, in a slightly different scenario, it is quite possible that the under-generalized transformer generated initially is the intended transformation. It is impossible to guess without semantic knowledge about the domain of the source code, which we are consciously keeping out-of-scope here.

However, we do have a *quiescence condition* on the environment and the synthesizer combined: when the user-dependent feedback stops changing (i.e., $\mathsf{Rew}_{\mathsf{U}}$ is fixed), the synthesized program should converge to a fixed one. Note that quiescence may be impossible under the situation where the user keeps adding more feedback or positive and negative examples. Due to the lack of strict correctness conditions, to ensure the quality of the programs and edits produced, we experimentally validate the techniques with a comprehensive evaluation (Section 6).

3.2 Background: Programming-by-Example for Code

Programming-by-Example (PBE) forms the basis of the techniques in our proposed solution. For a given input domain $\mathbb{I}$, output domain $\mathbb{O}$, and class of programs $\mathsf{Programs}$, a programming-by-example technique takes as input a set of examples $\{\mathsf{i}_0 \mapsto \mathsf{o}_0, \ldots, \mathsf{i}_n \mapsto \mathsf{o}_n\}$ and produces a program $\mathsf{P} : \mathbb{I} \longrightarrow \mathbb{O}$ such that $\mathsf{P}(\mathsf{i}_k) = \mathsf{o}_k$ for all $0 \leq k \leq n$. In our setting, we fix $\mathbb{I} = \mathbb{T}$ and $\mathbb{O} = \mathbb{T}$.

We use a slightly modified version of ReFazer [Rolim et al. 2017] as our programming-by-example engine. In ReFazer, the programs are drawn from the domain-specific language (DSL) shown in Figure 3. The programs are composed of guards and transformers. Guards are the conjunction of predicates over nodes of the AST. The nodes are identified using XPath like queries and the predicates test the label, attributes, or position of the nodes. Transformers are two types:

- Selections: A `select` returns a subtree of the input. The subtree is identified as the n^{th} node that satisfies a guard.
- Constructions: A `construct` returns a subtree that is built by specifying the kind of node, its attributes, and its children. The children may be constructed using several different operators. For example, the operator `InsertChild(select, pos, node)` selects a node (called parent) from the input and returns the parent's children with an additional node at the position pos.

We do not provide details on how ReFazer synthesizes programs given examples. However, one important aspect of the ReFazer synthesis algorithm is that it prefers selections over constructions, i.e., when a particular subtree of the output can be selected from the input AST, ReFazer returns a program with the selection. The reader is referred to [Polozov and Gulwani 2015; Rolim et al. 2017] for further details.

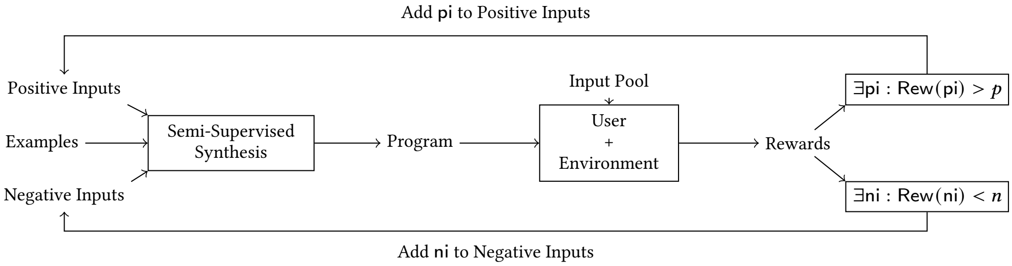


Fig. 4. Solution for the feedback-driven semi-supervised problem

Example 3.2. Let us revisit the edits in Example 3.1 . ReFazer synthesizes the following transformer: $\mathsf{X}_1.\mathsf{X}_2(\mathsf{X}_3,\mathsf{X}_4,\mathsf{X}_5,\mathsf{X}_6,\mathsf{X}_7) \not\mapsto \mathsf{DependencyResolveUtility}.\mathsf{X}_2(\mathsf{X}_1,\mathsf{X}_3,\mathsf{X}_4,\mathsf{X}_5,\mathsf{X}_6,\mathsf{X}_7)$. The ReFazer program that represents this transformer is

```
Tree(CallExpression, [], Cons(
  Tree(DotExpression, [], Cons(
    Tree(Identifier, [TextValue=DependencyResolveUtility], EmptyChildren),
    Cons(select1, EmptyChildren))),
  Cons(select2, select3)))
```

where, select1, select2, and select3 extract the fragments X_2, X_1, and $\mathsf{X}_3,\mathsf{X}_4,\mathsf{X}_5,\mathsf{X}_6,\mathsf{X}_7$ respectively. Each select is specified by a guard, for example, the select1 guard might be of the form $\mathtt{IsKind(Current, Identifier)} \wedge \mathtt{Attribute(Current, TextValue)} = \mathtt{ResolveDependency} \wedge \mathtt{IsKind(Parent, DotExpression)} \wedge \ldots$.

Over-generalization and Under-generalization. Input-output examples are inherently an under-specification of the intended program, and any programming-by-example technique needs to generalize inductively from the examples. Developers view false positives more unfavorably than false negatives—it causes them to lose trust in the tool [Bessey et al. 2010]. Hence, many synthesis techniques, including ReFazer, used in the source code transformation domain err on the side of under-generalization (for examples, see [Bader et al. 2019; Meng et al. 2013; Rolim et al. 2017]).

4 FEEDBACK-DRIVEN SEMI-SUPERVISED SYNTHESIS

We present our technique to address the feedback-driven semi-supervised synthesis problem. This solution approach is depicted in Figure 4 and works as follows:

- In each round, the feedback-driven problem with real number feedback is converted into an instance of the semi-supervised synthesis problem. We achieve this reduction by choosing thresholds p and n, with $\mathsf{PI} = \{\mathsf{i} \in \mathsf{InputPool} \mid \mathsf{Rew}_{n-1}(\mathsf{i}) > p\}$ and $\mathsf{NI} = \{\mathsf{i} \in \mathsf{InputPool} \mid \mathsf{Rew}_{n-1}(\mathsf{i}) < n\}$.
- The semi-supervised synthesis is solved using a standard (not semi-supervised) program synthesizer. To ensure that the synthesized program produces outputs on the additional positive inputs, we generate new examples by associating each additional positive input pi with an output po. This output is produced using a given example i ↦ o, and a combination of provenance analysis and anti-unification. Informally, we first associate each subtree s' of pi with an equivalent subtree s of i. Then, in o we replace each subtree generated from a subtree s of the input i, with a new subtree that is generated in a similar way but with s replaced by s'.

4.1 Semi-Supervised Synthesis

Algorithm 1 Semi-supervised synthesis

```
Input: Input-output examples $\mathsf{Examples} = \{\mathsf{i}_0 \mapsto \mathsf{o}_0, \ldots, \mathsf{i}_k \mapsto \mathsf{o}_k\}$
Input: Additional positive inputs $\mathsf{PI} = \{\mathsf{pi}_0, \ldots, \mathsf{pi}_n\}$
Input: Additional negative inputs $\mathsf{NI} = \{\mathsf{ni}_0, \ldots, \mathsf{ni}_m\}$
Output: Program $\mathsf{P}$
 1: $\mathsf{Inputs} \leftarrow \{\mathsf{i} \mid (\mathsf{i} \mapsto \mathsf{o}) \in \mathsf{Examples}\}$
 2: $\mathsf{P_{guard}} \leftarrow \mathsf{ReFazer_{guard}}(\mathsf{Inputs} \cup \mathsf{PI}, \mathsf{NI})$
 3: $\mathsf{P_{trans}} \leftarrow \mathsf{TransSynth}(\mathsf{Examples}, \mathsf{PI})$
 4: if $\mathsf{P_{guard}} = \bot \vee \mathsf{P_{trans}} = \bot$ then return $\bot$
 5: return $(\mathsf{P_{guard}}, \mathsf{P_{trans}})$
 6:
 7: function $\mathsf{TransSynth}(\mathsf{Examples}, \mathsf{PI})$
 8:     $\mathsf{P_{trans}} \leftarrow \mathsf{ReFazer_{trans}}(\mathsf{Examples})$
 9:     $\pi \leftarrow \mathsf{Provenance}(\mathsf{i}_0 \mapsto \mathsf{o}_0, \mathsf{P_{trans}})$
10:     $(\tau, \langle \sigma_0, \ldots, \sigma_k, \sigma'_0, \ldots, \sigma'_n \rangle) \leftarrow \bowtie_\pi \{\mathsf{i}_0, \ldots, \mathsf{i}_k, \mathsf{pi}_0, \ldots, \mathsf{pi}_n\}$
11:     if $\bot \in (\sigma_0, \ldots, \sigma_k, \sigma'_0, \ldots, \sigma'_n)$ then return $\bot$
12:     $\mathsf{AdditionalExamples} \leftarrow \{\mathsf{pi}_j \rightarrow \mathsf{Evaluate}^*(\mathsf{P_{trans}}, \mathsf{pi}, \mathsf{i}) \mid \mathsf{pi}_j \in \mathsf{PI}\}$
13:     return $\mathsf{ReFazer_{trans}}(\mathsf{Examples} \cup \mathsf{AdditionalExamples})$
```

Algorithm 1 depicts a procedure for the semi-supervised synthesis problem. In the procedure, we use $\mathsf{ReFazer_{guard}}$ and $\mathsf{ReFazer_{trans}}$ as oracles. Oracle $\mathsf{ReFazer_{guard}}$ takes positive inputs and negative inputs, and produces a guard that is true on the former and false on the latter. Oracle $\mathsf{ReFazer_{trans}}$ takes a set of examples and produces a transformer consistent with them.

The guard synthesis component of the algorithm (line 2) falls back to $\mathsf{ReFazer_{guard}}$. However, transformer synthesis is significantly more involved. First, using only $\mathsf{Examples}$, we synthesize a transformer program that is consistent with each example (line 8). Using this program, we extract *provenance information* (line 9) on what fragments of the example outputs are dependant on what fragments of the example inputs, and what sub-programs are used to transform the input fragments to the output fragments. Then, we use *anti-unification* (line 10) to determine which fragments of the example inputs are associated with which fragments of the additional positive inputs. Using the provenance and anti-unification data, we can now compute a candidate output for each additional positive input (line 12). Finally, we synthesize a transformer program from the original examples and the new examples obtained by associating each additional positive input with its candidate output. We explain these steps in detail below.

Provenance. The first step of transformer synthesis computes *provenance* information for each example. The provenance information is computed for `select` operations. Given a transformer program $\mathsf{P_{trans}}$, and an example $\mathsf{i} \mapsto \mathsf{o}$, the provenance information takes the form of $\mathsf{SP}_0 \leftarrow \mathsf{si}_0, \ldots, \mathsf{SP}_n \leftarrow \mathsf{si}_n$, where (a) each si_j is a subtree of i, and (b) each SP_j is a sub-program of $\mathsf{P_{trans}}$ that is a `select`, and SP_j produces the output si_j during the execution of $\mathsf{P_{trans}}(\mathsf{si})$. We call the subtrees si_j the *selected nodes* of the input i. Note that each SP_j may have multiple subtrees si_j and si'_j with $j \neq j'$ such that $\mathsf{SP}_j \leftarrow \mathsf{si}_j$ and $\mathsf{SP}_j \leftarrow \mathsf{si}'_j$. One such case is due to the `MapChildren` operator in Figure 3. The lambda function (produced by `transformer`) may have `select` programs that operate over all children of a given node.

Example 4.1. Consider the $\mathsf{P_{trans}}$ shown in Example 3.2 with the abbreviated example:
`repository.ResolveDependency(dependency1, args ...)` ↦
`DependencyResolverUtility.ResolveDependency(repository, dependency1, args ...)`

The provenance information is given by $\pi = \{$ `select1` $\leftarrow$ `ResolveDependency`, `select2` $\leftarrow$ `repository`, `select3` $\leftarrow$ `args`...$\}$.

Anti-Unification. The next step in the algorithm is to compute an anti-unification of inputs and additional positive inputs. Given two inputs i_1 and i_2, the *anti-unification* $\mathsf{i}_1 \bowtie \mathsf{i}_2$ is given by a pair $(\tau, \langle\sigma_1, \sigma_2\rangle)$ where:

- *template* τ, is an AST with labelled holes $\{\mathsf{h}_0, \dots, \mathsf{h}_n\}$, and
- two substitutions $\sigma_1, \sigma_2 : \{\mathsf{h}_0, \dots, \mathsf{h}_n\} \to \mathbb{T}$ such that $\sigma_1(\tau) = \mathsf{i}_1 \wedge \sigma_2(\tau) = \mathsf{i}_2$.

This definition can be generalized to more than two inputs. For arbitrary number of inputs, we use the notation $\bowtie\{\mathsf{i}_1, \dots, \mathsf{i}_n\}$. As is standard, we write anti-unification to mean the anti-unification that produces the most specific generalization.

Example 4.2. Consider the inputs i_1 = `if(score < limit)` and i_2 = `if(GetScore(run) < limit)`. Then the anti-unification $\bowtie\{\mathsf{i}_1, \mathsf{i}_2\}$ = `if(`h_0 `< limit)`, $\langle\{\mathsf{h}_0 \mapsto$ `score`$\}, \{\mathsf{h}_0 \mapsto$ `GetScore(run)`$\}\rangle$. It is more specific than any other generalization of i_1 and i_2, e.g., an anti-unification with template `if(`$\mathsf{h}_0 < \mathsf{h}_1$`)`.

We do not go into the details of the procedure for computing anti-unification but explain the procedure briefly. The procedure is a variant of anti-unification modulo associativity-unity (AU). First, we categorize all possible AST nodes into two different categories, based on the label:

- Fixed arity nodes: These are nodes that always have a fixed number of children. For example, Identifier always has 0 children, CallExpression always has 2 children (function and argument list), and PlusExpression always has 2 children.
- Variable arity nodes: These nodes can have different number of children. For example, ParameterList, Block, and ClassDeclaration. One key observation is that in the AST domain, the children of every variable arity node can be treated as a homogeneous list. That is, no position in the list has a special meaning: every child in a parameter list is a parameter. In contrast, the two children of CallExpression are functionally different.

Now, $\mathsf{i}_1 \bowtie \mathsf{i}_2$ is computed as follows:

- If the roots of i_1 and i_2 have different labels or attributes: $\mathsf{i}_1 \bowtie \mathsf{i}_2 = (\mathsf{h}, (\{\mathsf{h} \mapsto \mathsf{i}_1\}, \{\mathsf{h} \mapsto \mathsf{i}_2\}))$.
- If the root nodes of i_1 and i_2 have the same label label and attributes attrs, and if the nodes are fixed-arity: then $\mathsf{i}_1 \bowtie \mathsf{i}_2 = \mathsf{Tree}(\mathsf{label}, \mathsf{attrs}, \tau_1 \dots \tau_n), \langle\bigcup_i \sigma_1^i, \bigcup_i \sigma_2^i\rangle$ where (a) $\mathsf{Children}(\mathsf{i}_1) = \mathsf{i}_1^1, \dots, \mathsf{i}_1^n$ and $\mathsf{Children}(\mathsf{i}_2) = \mathsf{i}_2^1, \dots, \mathsf{i}_2^n$, and (b) for all $1 \leq j \leq n. \mathsf{i}_1^j \bowtie \mathsf{i}_2^j = (\tau_j, (\sigma_1^j, \sigma_2^j))$
- If the root nodes of i_1 and i_2 have the same label label and are variable arity nodes: Let the children of i_1 and i_2 be $\mathsf{i}_1^1, \dots, \mathsf{i}_1^n$ and $\mathsf{i}_2^1, \dots, \mathsf{i}_2^m$, respectively. Then, we compute two lists of node sequences $\mathsf{s}^0, \mathsf{d}_i^0, \mathsf{s}^1 \dots \mathsf{d}_i^k, \mathsf{s}^k$ for $i \in \{1, 2\}$ such that: (a) The concatenation $\mathsf{s}^0\mathsf{d}_i^0\mathsf{s}^1 \dots \mathsf{d}_i^k\mathsf{s}^k$ is equal to $\mathsf{i}_1^1, \dots, \mathsf{i}_1^n$ and $\mathsf{i}_2^1, \dots, \mathsf{i}_2^n$ for $i = 1$ and $i = 2$, respectively. Note that s^i and d_i^b are nodes that are shared and are different in the two lists, respectively. (b) the combined length of s_i^j is maximized. Note that some d_i^j may be the empty list nil which acts as the identity for the concatenation operation. Now, the anti-unification $\mathsf{i}_1 \bowtie \mathsf{i}_2 = (\mathsf{Tree}(\mathsf{label}, \mathsf{attrs}, \mathsf{s}^1\mathsf{h}_1 \dots \mathsf{s}^k), \langle\{\mathsf{h}_i \mapsto \mathsf{d}_1^i \mid 0 \leq i \leq k\}, \{\mathsf{h}_i \mapsto \mathsf{d}_2^i \mid 0 \leq i \leq k\}\rangle)$.

Remark 4.3. The anti-unification of two ASTs i_1 and i_2 is not uniquely defined. For example, let both i_1 and i_2 be argument lists with $\mathsf{i}_1 = (\mathtt{x}, \mathtt{x})$ and $\mathsf{i}_2 = (\mathtt{x})$ where `x` is a variable. Now, $\mathsf{i}_1 \bowtie \mathsf{i}_2$ is computed as per the third case above. As per the definition, we have two options for the result: (a) $((\mathtt{x}, \mathsf{h}), \langle\{\mathsf{h} \mapsto \mathtt{x}\}, \{\mathsf{h} \mapsto \mathsf{nil}\}\rangle$, or (b) $((\mathsf{h}, \mathtt{x}), \langle\{\mathsf{h} \mapsto \mathtt{x}\}, \{\mathsf{h} \mapsto \mathsf{nil}\}\rangle$. That is, it is unclear if the `x` in i_2 matches with the first or the second `x` in i_1. This issue can be resolved by using more advanced anti-unification techniques.

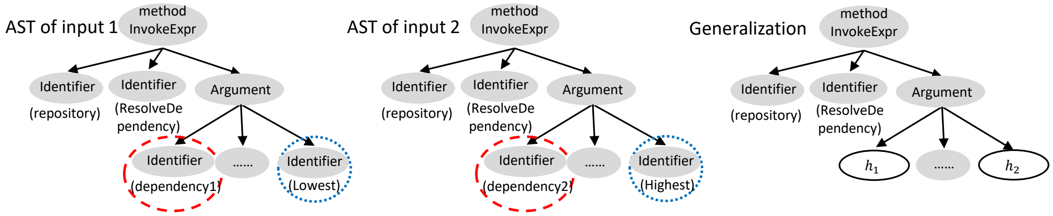


Fig. 5. The partial AST of two inputs shown in Figure 1a and 1c, and their anti-unification.

For our use case, we do not consider the general notion of anti-unification, but *anti-unification modulo provenance.* Consider inputs i_1 and i_2, and provenance information π derived from evaluation a transformation $\mathsf{P}_{\mathsf{trans}}$ on i. The anti-unification modulo provenance $\mathsf{i}_1 \bowtie_\pi \mathsf{i}_2$ is given by $(\tau, \langle\sigma_1, \sigma_2\rangle)$ where:

- $(\tau, \langle\sigma_1, \sigma_2\rangle)$ is an anti-unification of i_1 and i_2, i.e., $\sigma_1(\tau) = \mathsf{i}_1$ and $\sigma_2(\tau) = \mathsf{i}_2$; and
- For each substitution $(\mathsf{h} \mapsto \mathsf{si}) \in \sigma_1$, either (a) si is a selected node, i.e., $(\mathsf{SP} \leftarrow \mathsf{si}) \in \pi$ for some SP; or (b) si has no ancestors or descendants that are selected nodes. Note that this condition is only relevant for σ_1 as the provenance π is derived from evaluating a transformation on i.

The additional constraint on the substitutions makes anti-unification modulo provenance be undefined in certain cases (see Example 4.4).

Example 4.4. Consider the input i_1 = `score < limit` from the example `score < limit` ↦ `IsValid(score)` and the additional input i_2 = `GetScore(run) < limit`. Given i_1 and i_2, the anti-unification procedure generates substitutions $\sigma_1 = \{\mathsf{h} \mapsto$ `score`$\}$ and $\sigma_2 = \{\mathsf{h} \mapsto$ `GetScore(run)`$\}$ with the template h < `limit`. Given the input-output example and its corresponding transformation, the provenance procedure produces $\pi = \{\mathsf{SP} \leftarrow$ `score`$\}$ for some sub-program SP that is a `select` operation. Note that `score` in $\sigma_1 = \{\mathsf{h} \mapsto$ `score`$\}$ is a selected node in π, and thus, the anti-unification modulo π of i_1 and i_2 exists. Now, consider another input i_3 = `score == GetScore(run)`. Given i_1 and i_3, the anti-unification procedure generates substitutions $\sigma'_1 = \{\mathsf{h} \mapsto$ `score < limit`$\}$ and $\sigma'_3 = \{\mathsf{h} \mapsto$ `score == GetScore(run)`$\}$ with the template h. Here, the root node of i_1 is `LessThanExpression` and of i_3 is `EqualsExpression`: hence, the expressions cannot be unified further. In this case, the condition for the anti-unification modulo π does not hold, as the substitution $\mathsf{h} \mapsto \mathsf{i}_1$ returns the root node of i_1 which is not a selected node, but has a descendant that is a selected node. Thus, the anti-unification modulo π of i_1 and i_3 does not exist.

Intuitively, we are trying to match "important parts" (here, selected nodes) of i_1 with equivalent parts in i_2 and i_3. We can match the nodes `score` in i_1 and `GetScore(run)` in i_2 as they are represented by the same hole in the anti-unification, and thus, they are compatible. Conversely, we cannot match `score` in i_1 and `score` in i_3, because, even though they are equal, there is no hole in the anti-unification of i_1 and i_3 that maps to them. Thus, they are incompatible.

Completing the Procedure. Given the above anti-unification modulo provenance computation, we produce the potential outputs for all additional positive inputs PI. For producing these outputs, we use an evaluation process that uses an input i from an example and an additional input pi. This process is denoted as $\mathsf{Evaluate}^*(\mathsf{P}_{\mathsf{trans}}, \mathsf{pi}, \mathsf{i})$. Let σ and σ' be the substitutions for i and pi in the anti-unification modulo provenance, respectively. We evaluate $\mathsf{P}_{\mathsf{trans}}$ on pi as follows:

- For every sub-program SP of $\mathsf{P}_{\mathsf{trans}}$ which is a `select`, let $\mathsf{SP} \leftarrow \mathsf{si} \in \pi$. Then, the evaluation value is set to $\sigma'(\sigma^{-1}(\mathsf{si}))$.

- For every sub-program SP of $\mathsf{P}_{\mathsf{trans}}$ which is not a `select`, we evaluate the value by applying the top level operator on the evaluated values of the children, as usual.

Example 4.5. Consider the first input in Figure 1a and 1c, anti-unification generates $\sigma_1 = \{\mathsf{h}_1 \mapsto \texttt{dependency}, \mathsf{h}_2 \mapsto \texttt{Lowest}\}$ and $\sigma_2 = \{\mathsf{h}_1 \mapsto \texttt{dependency2}, \mathsf{h}_2 \mapsto \texttt{Highest}\}$. In order to produce an output for the additional positive input in 1c, we apply $\sigma_2(\sigma_1^{-1}(\mathsf{si}))$ to every $\mathsf{SP} \leftarrow \mathsf{si} \in \pi$. The elements of interest in π are: `select1` $\leftarrow$ `dependency` and `select2` $\leftarrow$ `Lowest` for some select sub-programs `select1` and `select2`. Now, we have $\sigma_2(\sigma_1^{-1}(\texttt{dependency})) = \texttt{dependency1}$ and $\sigma_2(\sigma_1^{-1}(\texttt{Lowest})) = \texttt{Highest}$. Using these values as the evaluation results of `select1` and `select2` and continuing evaluation, we end up with the output `DependancyResolverUtility.ResolveDependency(dependency1, ..., Highest)`.

Once we have the outputs for the additional positive inputs, we provide the given examples along with the new examples generated from additional positive inputs to the transformer synthesis component of ReFazer.

Theorem 4.6 (Soundness). Algorithm 1 is sound: if a program P is returned, then (a) $\forall \mathsf{i} \mapsto \mathsf{o} \in \mathsf{Examples}.\mathsf{P}(\mathsf{i}) = \mathsf{o}$, (b) $\forall \mathsf{pi} \in \mathsf{PI}.\mathsf{P}(\mathsf{pi}) \neq \bot$, and (c) $\forall \mathsf{ni} \in \mathsf{NI}.\mathsf{P}(\mathsf{ni}) = \bot$.

The proof follows from the use of $\mathsf{ReFazer}_{\mathsf{trans}}$ and $\mathsf{ReFazer}_{\mathsf{guard}}$ in lines 13 and 2, respectively. Note that, it is possible that the inferred output po for the additional positive input pi is incorrect. In this situation, the user can add a new input-output example (positive or negative) that has the same input that was incorrectly classified. We will ignore the additional input i if there exists an input from the input-output examples that is same as i.

Remark 4.7 (Completeness of Algorithm 1). Algorithm 1 is not complete, i.e., it may not return a program even when one satisfying all requirements exists. This is an intentional choice. Consider the case where Examples = {"`(temp − 32) * (5/9)`" ↦ "`FtoC(temp)`"}, PI = {"`x = x + 1;`"}, and NI = ∅. Here, the input of the example and the additional positive input are not logically related. However, there exists a program that is correct, i.e., the program that returns the constant tree "`FtoC(temp)`". In any practical scenario, this constant program is very unlikely to be the intended program. Hence, we explicitly make the choice of incompleteness.

4.2 Feedback-Driven Semi-Supervised Synthesis

Algorithm 2 presents a procedure for the feedback-driven semi-supervised synthesis problem that closely follows Figure 4. It takes the following as input: (a) A feedback oracle Feedback that represents the user and the environment. The feedback oracle takes as input a program P, a set of examples Examples, an input pool InputPool, positive inputs PI, and negative inputs NI, and produces a reward function $\mathsf{Rew} : \mathsf{InputPool} \to [-\infty, \infty]$. Informally, the feedback oracle checks the whole state of the process, and produces rewards for inputs from the pool. (b) A semi-supervised synthesis procedure SynthesisEngine depicted in Algorithm 1. (c) An input pool, an initial non-empty set of examples, a set of positive inputs, and a set of negative inputs.

In addition, the algorithm uses the thresholds p and n to determine if an input from the input pool should be added to either the positive or negative inputs. These thresholds are dependant on the application scenario and the Feedback oracle. In Section 5, we present three different application scenarios and the choice of p and n for them. For the Feedback oracle, we present two different oracles $\mathsf{Feedback}_{\mathsf{user}}$ and $\mathsf{Feedback}_{\mathsf{auto}}$. In the application scenarios, these oracles are combined in different ways to obtain application specific feedback oracles.

User-Driven Feedback Oracle. The *user-driven feedback oracle* $\mathsf{Feedback}_{\mathsf{user}}$ represents the user of the application. In different interfaces, the feedback from the user can take different forms, each

Algorithm 2 Feedback-driven semi-supervised synthesis

```
Require: Feedback oracle Feedback : $\mathcal{P} \times (\mathbb{T} \nrightarrow \mathbb{T}) \times 2^{\mathbb{T}} \times 2^{\mathbb{T}} \times 2^{\mathbb{T}} \to (\mathbb{T} \to [-\infty, \infty])$.
Require: Semi-supervised synthesis engine SynthesisEngine.
Require: Pool of available inputs InputPool.
Require: Initial examples Examples, positive inputs PI, and negative inputs NI.
Require: Thresholds $p, n \in \mathbb{R}$.
while true do
    P ← SynthesisEngine(Examples, PI, NI)
    Notify user of current suggestions: $\{i \mapsto o \mid i \in \mathsf{InputPool} \wedge o = P(i) \wedge o \neq \bot\}$
    Rew ← Feedback(P, Examples, InputPool, PI, NI)
    PI′ ← $\{i \in \mathsf{InputPool} \mid \mathsf{Rew}(i) > p\}$
    NI′ ← $\{i \in \mathsf{InputPool} \mid \mathsf{Rew}(i) < n\}$
    if * then
        PI ← PI ∪ pi′ where pi′ is an arbitrary input from PI′
    else
        NI ← NI ∪ ni′ where ni′ is an arbitrary input from NI′
```

of which can be converted to a reward function $\mathsf{Rew}_{\mathsf{U}} : \mathsf{InputPool} \to [-\infty, +\infty]$. We have the following two cases (Section 5):

- The user explicitly provides new positive inputs PI′ and negative inputs NI′. We convert this feedback into the reward function $\mathsf{Rew}_{\mathsf{U}}$ by setting $\forall \mathsf{pi} \in \mathsf{PI}'.\mathsf{Rew}_{\mathsf{U}}(\mathsf{pi}) = +\infty$, $\forall \mathsf{ni} \in \mathsf{NI}'.\mathsf{Rew}_{\mathsf{U}}(\mathsf{ni}) = -\infty$, and $\mathsf{Rew}_{\mathsf{U}}(\mathsf{i}) = 0$ for all other inputs in InputPool.
- The user provides a set of candidate positive inputs PI^* with the intent that the transformation should apply to one of these candidate positive inputs. For example, a set of candidate positive inputs could be a set of ASTs that contain the cursor location in a file. We give a constant reward to all the nodes in PI^*, i.e., we have $\forall \mathsf{pi} \in \mathsf{PI}^*.\mathsf{Rew}_{\mathsf{U}}(\mathsf{pi}) = C$ where $0 < C < +\infty$. In our implementation, we set C as 2.

With richer user interfaces, we could consider more complex forms of $\mathsf{Feedback}_{\mathsf{user}}$ oracle.

Fully Automated Feedback Oracle. The *fully automated feedback oracle* $\mathsf{Feedback}_{\mathsf{auto}}$ represents the environment the synthesizer is operating in. It can include a number of independent components only restricted by the available tools in the environment the synthesizer is running in. For example, if a synthesizer is running inside an IDE, the oracle could use the compiler or the version control history. Algorithm 3 presents a basic oracle that reuses the provenance and anti-unification computation from the semi-supervised synthesis engine, and, uses the scoring function Score on guards and a bound $\mathsf{threshold}_g$ on scores. The scoring function and bound we use are the same as in BluePencil [Miltner et al. 2019], which in turn takes the scoring function from [Rolim et al. 2017]. In practice, the feedback loop in Algorithm 2 can be optimized by sharing the provenance computation and anti-unification across the synthesis engine and the $\mathsf{Feedback}_{\mathsf{auto}}$ oracle.

Algorithm 3 works as follows. For each candidate (positive or negative) additional input i in the input pool:

(a) If the program P on i produces an output and that output cannot be compiled, reward is $-\infty$ (line 5). Though compilation can be expensive, in practice, IDEs allow for efficient incremental compilation. Further, this step is not as expensive as P typically does not produce an output, i.e. $\mathsf{P}_{\mathsf{guard}}(\mathsf{i}) = \mathsf{false}$, for most $\mathsf{i} \in \mathsf{InputPool}$.

(b) Otherwise, we synthesize a guard that matches the examples and positive inputs along with the candidate input i using $\mathsf{ReFazer}_{\mathsf{guard}}$. Similar to BluePencil [Miltner et al. 2019], we bound

Algorithm 3 The fully automated feedback oracle $\mathsf{Feedback}_{\mathsf{user}}$

```
Require: Compiler Compiler : $t \to \mathbb{B}$ or $\bot$ if compiler is not available
Require: Distance metric Distance : $\mathbb{T} \times \mathbb{T} \to \mathbb{R}^{\geq 0}$
Require: Program $\mathsf{P} = (\mathsf{P}_{\mathsf{guard}}, \mathsf{P}_{\mathsf{trans}})$
Require: Examples Examples : $\mathbb{T} \times \mathbb{T}$
Require: Input pool InputPool
Require: Positive Examples PI, Negative Examples NI
Ensure: Rewards function $\mathsf{Rew}_{\mathsf{E}} : \mathsf{InputPool} \not\to [-\infty, +\infty]$
 1: $\mathsf{i}^* \mapsto \mathsf{o}^* \leftarrow$ arbitrary example in Examples
 2: $\pi \leftarrow \mathsf{Provenance}(\mathsf{i}^* \mapsto \mathsf{o}^*, \mathsf{P}_{\mathsf{trans}})$
 3: $\mathsf{Rew}_{\mathsf{E}} \leftarrow \emptyset$
 4: for all $\mathsf{i} \in \mathsf{InputPool}$ do
 5:     if $\mathsf{P}(\mathsf{i}) \neq \bot \land \mathsf{Compiler} \neq \bot \land \mathsf{Compiler}(\mathsf{P}(\mathsf{i})) = \mathsf{false}$ then
 6:         $\mathsf{Rew}_{\mathsf{E}} \leftarrow \mathsf{Rew}_{\mathsf{E}} \cup \{\mathsf{i} \mapsto -\infty\}$
 7:         continue
 8:     $\mathsf{guard} \leftarrow \mathsf{ReFazer}_{\mathsf{guard}}(\{\mathsf{i} \mid \mathsf{i} \mapsto \mathsf{o} \in \mathsf{Examples}\} \cup \mathsf{PI} \cup \{\mathsf{i}\}, \mathsf{NI})$
 9:     if $\mathsf{Score(guard)} < \mathsf{threshold}_g$ then
10:         $\mathsf{Rew}_{\mathsf{E}} \leftarrow \mathsf{Rew}_{\mathsf{E}} \cup \{\mathsf{i} \mapsto -\infty\}$
11:         continue
12:     $\mathsf{d} \leftarrow 1 - \mathsf{Distance}(\mathsf{i}, \mathsf{i}^*)$
13:     $\mathsf{Rew}_{\mathsf{E}} \leftarrow \mathsf{Rew}_{\mathsf{E}} \cup \{\mathsf{i} \mapsto \mathsf{d}\}$
14: return $\mathsf{Rew}_{\mathsf{E}}$
```

the score of the guard with a threshold to avoid overly general guards, which are almost never the intended one (line 9).

(c) Otherwise, we compute the distance between the candidate input i and an example input i^*, using a Distance function, i.e. $\mathsf{Rew}_{\mathsf{E}} = 1 - \mathsf{Distance}(\mathsf{i}, \mathsf{i}^*)$, where $\mathsf{Distance}(\mathsf{i}, \mathsf{i}^*) \in [0, 1]$ (line 12). The Distance function is explained in detail below.

The Distance Function. Consider an input i^* that comes from an example $\mathsf{i}^* \mapsto \mathsf{o}^*$, and a candidate additional input i. Intuitively, we want to give a high reward if i is similar to i^*. However, we need a more involved notion of similarity than standard clone detection techniques.

Example 4.8. Consider the example `if(score < limit)` ↦ `if(IsValid(score))` and the candidate additional input `if(GetScore(run) < limit)`. A tree-based clone-detection technique would not classify the above two inputs as clones given the high difference between `score` and `GetScore(run)`. However, as we described in Example 4.4, the anti-unification modulo π of these inputs tells us that (i) `score` is a relevant part of the input since it also appears in the output, and (ii) `score` and `GetScore(run)` are compatible since there is a hole in the anti-unification that maps to these nodes.

Given that we already have this information about the compatibility of these subtrees, we "relax" the tree distance comparison between these two inputs. Rather than comparing the concrete subtrees, we abstract them using a technique called d-caps [Evans et al. 2009; Nguyen et al. 2013]. For a $d \geq 0$, the d-cap of a node replaces all the sub-nodes at depth d with holes. For instance, when $d = 1$, instead of comparing `score` and `GetScore(run)`, we compare the nodes (with no children) Identifier and `CallExpression`, which are their corresponding root nodes. Note that expression `score` is shorthand for a node with label Identifier, attributes $\{\mathsf{TextValue} \mapsto \mathsf{score}\}$, and no children. Both the subtrees have been truncated to a depth of 1. This "loosens" the comparison between these nodes, and returns a smaller difference value.

On the other hand, consider the candidate additional input `if(score > UnrelatedFunction())`. Now, the difference between the two inputs is due to `< limit` and `> SomeUnrelatedFunction()`. These two fragments are not directly used in the output, and thus we cannot rely on the anti-unification modulo π to assess their compatibility. Hence, it is essential that we include this particular difference in the computation of distance.

Concretely, our Distance function represents the d-cap replaced input as numerical vectors and uses the Euclidean distance between these vectors to represent the distance between the trees, similar to Deckard [Jiang et al. 2007], a clone detection technique. The distance between the two inputs i_1 and i_2 can then be formally defined as follows:

$$\begin{aligned} \mathsf{Distance}(\mathsf{i}_1, \mathsf{i}_2) = \mathsf{CloneDetection}(\sigma_1^\dagger(\tau), \sigma_2^\dagger(\tau)) \text{ where} \\ (\tau, \langle \sigma_1, \sigma_2 \rangle) = \mathsf{i}_1 \bowtie_\pi \mathsf{i}_2 \\ \sigma_1^\dagger, \sigma_2^\dagger = \mathsf{DCapModuloProvenance}(\sigma_1, \sigma_2, \pi) \end{aligned}$$

Here, DCapModuloProvenance replaces each substitution for a selected subtree with its *d-cap*. Formally, $\sigma_i^\dagger(\mathsf{h})$ is equal to: (a) the d-cap of $\sigma_i(\mathsf{h})$ if $\sigma_i(\mathsf{h})$ is a selected node, and (b) $\sigma_i(\mathsf{h})$ otherwise.

5 APPLICATIONS OF SEMI-SUPERVISED SYNTHESIS

In this section, we present three practical applications of semi-supervised synthesis in the domain of C# program transformations. They allow different types of feedback to produce additional positive inputs to the semi-supervised synthesizer. To implement the semi-supervised synthesis algorithm (Algorithm 1), we leverage the Transformation.Tree API available in the PROSE SDK as a concrete implementation of ReFazer. Additionally, in all applications, we use all the AST nodes available in the source code file as inputs for the input pool. In our implementation, we use untyped ASTs, i.e., each node in the AST does not have the type of the corresponding expression as an attribute. While our techniques are able to handle typed ASTs, performing type inference on every edit can incur performance penalties.

5.1 ReFazer*: User-Provided Feedback about Additional Inputs

ReFazer* uses the user-driven feedback oracle to identify positive inputs to the semi-supervised synthesizer. The target for ReFazer* is applications where a developer is providing examples manually. To illustrate this application, consider our motivating example shown in Figure 1. For the first false negative (Figure 1c), instead of manually performing the edit to give another example, the developer can provide feedback to the system by indicating that the location (text selection representing the input AST) should have been modified. ReFazer* uses the feedback to create a positive input and generalize the transformation. After that, ReFazer* produces suggestions to two out of the three false negatives. The developer can follow the same process to fix the other false negative. In terms of the feedback oracles from the previous section, $\mathsf{Feedback}_{\mathsf{user}}$ returns a reward function Rew_U that is $+\infty$ on the additional positive input the developer has provided, and 0 everywhere else. Further, we pick the thresholds p and n to both have the value 0. Similarly, if ReFazer* produces a false positive on some location, developers could provide feedback to the system by indicating that this location (text selection and press predefined shortcut) should not be modified. With this feedback, $\mathsf{Feedback}_{\mathsf{user}}$ returns a reward $-\infty$ on the additional negative input provided by developers. Correspondingly, ReFazer* will refine the synthesized transformation with additional examples to avoid generating similar false positives.

ReFazer* requires the developer to enter a special mode to provide examples and feedback to the system. While this interaction gives more control to the developer, it may also prevent developers from using it due to discoverability problems [Miltner et al. 2019]. Next, we describe two other *modeless* applications of our technique that do not require explicitly providing examples and feedback.

5.2 BluePencil$_{\text{cur}}$: Semi-automated Feedback Based on Cursor Position

For our second application, we instantiated the BluePencil algorithm [Miltner et al. 2019] using our semi-supervised synthesizer as the PBE synthesizer. BluePencil works in the background of an editor. While the developer edits the code, the system infers examples of repetitive edits from the history of edits, and it uses a synthesizer to learn program transformations for these edits. The original algorithm does not consider sets of input-output examples of size one, as they do not indicate repetitive changes. We modified this constraint to allow the system to use BluePencil$_{\text{cur}}$ to learn transformations from just one example and one additional positive input.

To enable the completely modeless interaction, BluePencil$_{\text{cur}}$ uses both user-driven and fully automated oracles to produce feedback. The former leverages the cursor position to collect implicit feedback from the developer. Note that the developer is not actively providing feedback—it is completely transparent to the developer, and is inferred automatically. Intuitively, the cursor suggests that the developer is interested in that part of the code and may want to edit it.

However, the cursor location is very ambiguous: the subtree the developer is likely to edit can be any subtree that contains the cursor location. Consider the false negative shown in Figure 1c. Suppose the developer places the cursor location at the beginning of the line. There are many subtrees that include this location, including the ones corresponding to the following code fragments: `repository` and `repository.ResolveDependency(...)`. The latter is the input that should be classified as a positive input. The $\mathsf{Feedback}_{\mathsf{user}}$ oracle returns a reward function that gives a positive score ($\mathsf{Rew}_{\mathsf{U}}$) to all subtrees that include the position defined by the cursor. We also use feedback from the $\mathsf{Feedback}_{\mathsf{auto}}$ oracle described in Section 4.2 to further disambiguate the cursor location. Intuitively, $\mathsf{Feedback}_{\mathsf{auto}}$ will provide positive rewards ($\mathsf{Rew}_{\mathsf{E}}$) to the nodes that are "similar" to the example inputs. Finally, we regard inputs with $\mathsf{Rew}_{\mathsf{U}}(i) * \mathsf{Rew}_{\mathsf{E}}(i) > p$ as positive inputs and inputs with $\mathsf{Rew}_{\mathsf{U}}(i) * \mathsf{Rew}_{\mathsf{E}}(i) < n$ as negative inputs.

We implement BluePencil$_{\text{cur}}$ as a Visual Studio extension. Figure 2 shows the extension in action. As soon as the developer places the cursor in the location related to the false negative, BluePencil$_{\text{cur}}$ uses the semi-supervised feedback synthesis to improve the transformation. The new transformation produces an *auto-completion* suggestion for the current location (see Figure 2). In this setting, we are using the user-driven feedback and the automated feedback to more precisely pick the additional positive input. However, there are many settings where it is infeasible to obtain any feedback from the user. We discuss this case in the next section.

5.3 BluePencil$_{\text{auto}}$: Fully Automated Feedback Based on all Inputs in the Source Code

Our last application (BluePencil$_{\text{auto}}$) uses fully automated feedback to identify positive inputs without any explicit or implicit feedback from developers. The motivation for this application is that the developers may not be aware of all locations that must be changed or they may want to apply the edits in bulk. We also implemented BluePencil$_{\text{auto}}$ on top of BluePencil. We restricted this application to synthesis tasks that have at least two input-output examples.

Consider again our motivating example (Figure 1). As soon as the developer finishes the first two edits (Figure 1a), BluePencil$_{\text{auto}}$ automatically identifies the inputs in Figure 1c as positive inputs and synthesizes the correct transformation. Now, if the developer is unaware of the other locations, the tool still produces suggestions at these places. These suggestions may then be used to automatically prompt the developer to make these additional edits. Another scenario is as follows: after the two edits, the developer creates a *pull request*. The tool can now be run as an automated reviewer (see, for example, [Bader et al. 2019]) to suggest changes to the pull request.

6 EVALUATION

In this section, we present our evaluation of the proposed approach in terms of effectiveness and efficiency. In particular, we evaluate our technique with respect to the following research questions:

(RQ1) What is the effectiveness of ReFazer* in generating correct code transformations? We hypothesize that user-provided positive inputs should help our synthesis engine learn better transformations. We evaluate the quality of the synthesized transformation with and without additional positive inputs by measuring the number of false positives (incorrect suggestions) and false negatives (missing suggestions) produced.

(RQ2) What is the effectiveness of the reward calculation function? The reward calculation function needs to precisely identify valid additional inputs to avoid generating many false positives or false negatives. We evaluate our reward calculation function by comparing it with two baseline approaches: *no validation* and *clone detection*.

(RQ3) Given a cursor location, what is the effectiveness and efficiency of BluePencil$_{\text{cur}}$? BluePencil$_{\text{cur}}$ should generate edit suggestions at the cursor location efficiently enough to be usable as an auto-completion feature in an IDE, while still maintaining the quality of suggestions. Given cursor locations, we measure the number of false positives and negatives produced by BluePencil$_{\text{cur}}$, and the time taken to produce the suggestions.

(RQ4) How do BluePencil$_{\text{cur}}$ and BluePencil$_{\text{auto}}$ compare to BluePencil? BluePencil$_{\text{cur}}$ and BluePencil$_{\text{auto}}$ are both built on top of BluePencil, and they aim at reducing the number of examples developers need to provide. By simulating a developer performing repetitive edits using these tools, we compare how much information (examples and locations) is required by each one of them.

6.1 Benchmark Suite

We collected 86 occurrences of real life code editing sessions containing repetitive edits. These scenarios were collected from developers at Microsoft spanning multiple teams during the internal testing phase of the Microsoft Visual Studio IntelliCode suggestions feature (BluePencil).

Each session consists of a list of program versions representing the history of the program content as the user makes edits. For each session, we manually generated the ground truth data containing the *number of repetitive edits*, the *version ids* before and after each repetitive edit, and the *locations* and *content change* for each repetitive edit. Each editing session contains one or multiple sequences of repetitive edit transformations, with each sequence containing at least two repetitive edits. Each session also contained noise, i.e., edits that are not a part of any repetitive sequence. Figure 6 shows the number of repetitive edits in different program editing sessions, where the x-axis presents the number of repetitive edits and y-axis gives the number of editing sessions. For instance, there are 25 (around 30%) editing sessions with 2 repetitive edits. This high percentage also motivates the need for a technique that automates edits with fewer examples, ideally 1 example. Techniques such as BluePencil that require at least two examples cannot generate any suggestions for cases with just 2 repetitive edits in the session. The average number of repetitive edits is 4.07 while the largest number is 16. The benchmark suite contains a variety of edits, from small edits that change only a single program statement to large edits that modify code blocks.

All the experiments were conducted on a machine equipped with Inter Core i7-8700T CPU @ 2.4GHz, 32GB memory running 64-bit Windows 10 Enterprise.

6.2 Effectiveness of ReFazer*

In the scenario where a developer manually indicates an additional positive input for a repetitive transformation, we evaluate the effectiveness of ReFazer* by measuring its precision and recall

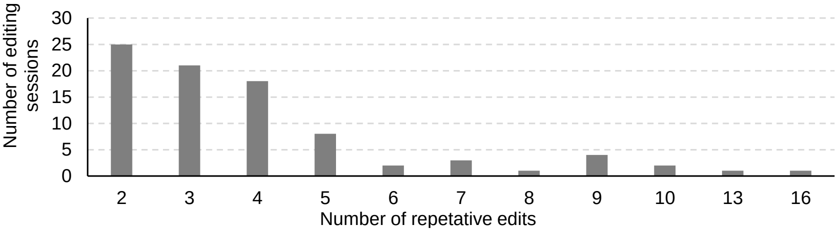


Fig. 6. The distribution of number of repetitive edits across the programs

Table 1. The effectiveness of semi-supervised synthesis.

Examples (N)	Session	Edit	Scenario	ReFazer		ReFazer*	
				Precision	Recall	Precision	Recall
One	86	350	1400	100.00%	26.71%	96.01%	100.00%
Two	61	300	3664	99.65%	77.26%	98.58%	99.94%
Three	40	237	7578	99.88%	89.10%	99.72%	99.99%

in generating correct suggestions. In this evaluation, we use ReFazer [Rolim et al. 2017] as our baseline.

Experimental Setup. In each program editing session, we first manually extract all the repetitive edits. For a session with M repetitive edits, we provide N edits as examples for the synthesis engine, and the remaining repetitive edits in this session are used for testing. We set $N < M$ to ensure there is at least one edit that can be used for testing, further, we limit N up to three. Considering that users could perform the repetitive edits in any order, we consider all combinations when choosing the examples. For instance, for a session with three repetitive edits (e_1, e_2, e_3), the users could manually complete e_1 and ReFazer* automates e_2 and e_3. The user could also complete e_3, and ReFazer* automates e_1 and e_2. Different edits contain slightly different information: the result of the synthesizer depends not only on the number of examples but also on which examples were used. We try all combinations of N examples to avoid any bias introduced by picking a particular order. For an editing session with M repetitive edits, there are $C(M, N)$ combinations when choosing the N examples. For instance, for a program edit session with four repetitive edits, if two edits are provided to the PBE engine as examples, there are $C(4, 2) = 6$ combinations. Given a combination of N examples to the PBE engine, we then create a set of testing scenarios where the N edits are provided to PBE engine as examples, and one of the $M - N$ other edits is used for testing. Therefore, for an editing session with M repetitive edits, we create $C(M, N) * (M - N)$ scenarios. In each test, ReFazer* also takes the input from testing edit as additional positive input. We then compare the output of the synthesized transformation on the test input against the test output. We calculate the precision and recall of ReFazer and ReFazer* by measuring the number of false positives, false negatives, and true positives produced in all the scenarios.

Experimental Parameters. In this experiment, we set $\mathsf{Rew}_\mathsf{U}(\mathsf{pi}) = +\infty$ for the user-provided positive input $\mathsf{pi} \in \mathsf{PI}$ and $\mathsf{Rew}_\mathsf{U}(\mathsf{ni}) = -\infty$ for the user-provided negative input $\mathsf{ni} \in \mathsf{NI}$. Further, we set both p and n in Algorithm 2 as 0.

Evaluation Results. Table 1 presents our evaluation results of traditional ReFazer and ReFazer*. The first column displays the number of examples provided to PBE engine, while the Session column shows the number of program editing sessions. Edit and Scenario columns display the number of edits and scenarios, respectively. The more examples the PBE engine takes, the more scenarios we create because there are more combinations when choosing examples. By comparing the different number of examples, ReFazer produces much better results (recall) with more examples (from 26.71% with one example to 89.10% with three examples). This is because the synthesis engine can learn how to generalize the transformation with more examples. The precision is always high because ReFazer always learns the most specific transformation which is unlikely to produce false positives. However, too specific transformations easily result in false negatives. Especially, the recall with one example is just 26.71%, which highlights the challenges of synthesizing a high-quality transformation with fewer examples. In contrast, ReFazer* significantly improves the recall regardless of the number of examples, while maintaining the high precision (slightly lower). ReFazer* can generate better results because the additional input helps synthesize a more suitably generalized transformation. Specifically, we achieve 100% recall and >96% precision with only one example, which can release the burden of users from providing multiple repetitive edit examples. Compared to ReFazer, we generate a few more false positives. The nature of these additional false positives is discussed in Section 6.6.

> ReFazer* significantly improves the recall of ReFazer while retaining the high precision in generating correct suggestions. Even by taking one example as input, ReFazer* achieves more than 96% precision and 100% recall.

6.3 Effectiveness of Reward Calculation Function

Our second experiment evaluates the effectiveness of the proposed reward calculation function. The reward calculation function determines whether a node is an additional positive or negative input for the feedback system. In this section, we evaluate its effectiveness in identifying additional positive inputs by comparing with two baseline approaches: *No validation* and *clone detection*.

- *No validation*: This baseline regards any node as an additional positive input. Hence, we set $\mathsf{Rew}(\mathsf{i}) = +\infty$ for all nodes in the input pool.
- *Clone detection*: Given an edit $\mathsf{i}^* \mapsto o^*$ and one additional node i, we determine whether i is an positive additional input by calculating the normalized distance between i^* and i using clone detection techniques, i.e. $\mathsf{Rew}(\mathsf{i}) = 1 - \mathsf{CloneDetection}(\mathsf{i}, \mathsf{i}^*)$. Here, we use the approach proposed by Jiang et al. [2007] without the use of the d-cap modulo provenance from Section 4.2.
- *Reward function based on Distance*: Given edit $\mathsf{i}^* \mapsto o^*$ and additional node i, we use our proposed approach in Algorithm 3 and Section 4.2 to calculate the reward score for i.

Experimental Setup. In each program editing session, we select the first edit as the example $\mathsf{i}^* \mapsto o^*$ for the PBE engine. We then create a set of additional inputs to test whether the techniques above can correctly classify each input i in this set as positive or negative. To create this set, we select the inputs of the remaining edits as positive inputs pNodes and all the remaining subtrees from the document that should not be transformed by the synthesized transformation as negative inputs nNodes. We measure the false positives and negatives produced on both pNodes and nNodes by the different approaches.

Experimental Parameters. In this experiment, we set p and n in Algorithm 2 as 0.7 and 0.1, respectively. Specifically, we regard input i as a positive input if $\mathsf{Rew}(\mathsf{i}) > 0.7$ and a negative input if $\mathsf{Rew}(\mathsf{i}) < 0.1$. Further, we set $d = 2$ for d-cap replacement (section 4.2).

Evaluation Results. Table 2 shows the evaluation results. By regarding any node as an additional positive input, the synthesis engine can successfully generate suggestions for many of them. However, it also generates a large number of false positives (9055), which demonstrates the importance

Table 2. The effectiveness of the reward calculation function.

Sessions	# pNodes	# nNodes	No validation		Clone detection		Reward function	
			# false positive	# false negative	# false positive	# false negative	# false positive	# false negative
86	265	243417	9055	7	8	111	11	14

Table 3. The effectiveness of BluePencil$_{\text{cur}}$ when given the history edit trace and the cursor location.

Scenarios	Suggestion	False Positive	False Negative	Precision	Recall	Time (ms)
295	291	1	3	99.66%	98.98%	51.83 (avg)

of the additional input validation. If we validate the additional input using existing clone detection (Column Clone detection), the false positive rate is significantly reduced. However, it introduces more false negatives because the clone detection is too strict when comparing two inputs as shown in Section 4.2. Considering the fact that we fail to generate suggestions on more than 40% (111 out of 265) of pNodes, the clone detection technique is also not acceptable. The last two columns show the evaluation result of our reward calculation function. We also significantly reduce the number of false positives and we do not introduce too many false negatives. Our reward calculation function results in 3 more false positives than clone detection. The underlying reason will be analyzed in the discussion section.

> The proposed additional input validation can help reduce false positives. Further, it also generates fewer false negatives than existing clone detection techniques.

6.4 The Effectiveness and Efficiency of BluePencil$_{\text{cur}}$

To evaluate the effectiveness and efficiency of BluePencil$_{\text{cur}}$, we measure the false positive and false negatives produced at the cursor location by simulating the program editing process of developers.

Experimental Setup. Recall that all the program versions are recorded in form of $\{v_1, v_2, v_3 \ldots v_i \ldots v_n\}$ on each program editing session. We could easily reproduce the editing steps by going through all the history versions one by one. From the second edit in each editing session (users need to manually complete the first edit), we feed the history versions before edit e_i and the edit location of e_i to BluePencil$_{\text{cur}}$. The history versions include at least one repetitive edit (e.g. e_1) and some irrelevant edits (noise). We randomly select a location from the range of edit location to simulate the cursor location (user might invoke synthesis at any location within the range of edit). We use the same experimental parameters as Section 6.3.

Evaluation Results. Table 3 shows our evaluation result. Scenarios presents the number of scenarios. In each scenario, one set of history versions and one cursor location are provided to the engine. Our evaluation results show that our engine only generates one false positive and three false negatives on all the scenarios. In other words, we achieve 99.66% precision and 98.98% recall.

Meanwhile, BluePencil$_{\text{cur}}$ should be fast enough to ensure that the suggestion can be generated at run-time. Therefore, we also evaluate the efficiency of BluePencil$_{\text{cur}}$ by measuring the time to generate each suggestion. Time describes the averaged time to generate edit suggestions. Our engine produces one suggestion in 51.8ms on average, and up to 441ms. At the cursor location, we believe generating suggestions in less than 0.44 seconds is acceptable.

> Given one set of history versions and one cursor location, BluePencil$_{\text{cur}}$ achieves around 99% precision and recall in generating correct suggestions. Meanwhile, it just takes 51.8 milliseconds on average to generate one suggestion.

6.5 A Comparison to BluePencil

In this section, we present an experiment that simulates a developer performing repetitive edits in two different settings.

- *Setting 1*: The developer uses BluePencil to complete the task.
- *Setting 2*: BluePencil$_{\text{cur}}$ and BluePencil$_{\text{auto}}$, which are built on top of BluePencil, are enabled and they assist the developer to complete the task.

The goal of this experiment is to compare the amount of information, in the form of examples and locations, that a developer must provide to complete a task when supported by these tools.

Experimental Setup. To simulate Setting 1, given an edit session that contains edits $\{e_1, e_2, ..., e_n\}$, we iteratively add each edit e_i as an example to BluePencil. At each iteration, we check the suggestions produced by BluePencil. If it produces a suggestion to automate an edit e_j, such that $j > i$, we remove this edit from the set of available edits. At the end of the simulation, we have the total number of examples #examples provided by the developer and the number of edits #suggestions that were automated by BluePencil. For instance, consider the scenario showed in Figure 1, where the developer performed seven repetitive edits. After providing e_1 and e_2 (Figure 1a) as examples to BluePencil, it produces the suggestions to automate e_3 and e_4 (Figure 1b). The three edits left are the ones that were applied to the locations shown in Figure 1c. We provide e_5 and it produces a suggestion to e_6. Finally, we provide e_7, the last edit. In total, we simulated the developer providing 4 examples (i.e., #examples = 4) and the BluePencil automating 3 edits (i.e., #suggestions).

Setting 2 is similar to Setting 1 but instead of simulating the developer interaction just with BluePencil, we add BluePencil$_{\text{cur}}$ and BluePencil$_{\text{auto}}$. Now, at each iteration after the first, before providing e_i, we first provide an arbitrary cursor location within the location of e_i. Only if BluePencil$_{\text{cur}}$ cannot produce the suggestion to automate e_i, we provide the full example. This process simulates a developer first navigating to the location of e_i and then performing the edit. If BluePencil$_{\text{cur}}$ is able to produce a suggestion for e_i as soon as the developer navigates to the location of e_i, it is counted towards the number of locations #locs. Otherwise, the developer has to manually perform this edit, and e_i is counted towards the number of examples #examples.

Further, we also enable BluePencil$_{\text{auto}}$ to automatically find additional inputs. For instance, back to our running scenario, after providing the first edit in Figure 1a as an example, we provide a cursor location within the second edit. Using this example and location, BluePencil$_{\text{cur}}$ produces suggestions for e_2, e_3, and e_4. Additionally, BluePencil$_{\text{auto}}$ produces suggestions for e_5, e_6, and e_7. Note that BluePencil$_{\text{auto}}$ requires at least two examples (see Section 5), and thus will not produce any suggestions until the user provides at least one edit and one cursor location. In this simulation, the developer provided one example (i.e., #examples = 1) and one cursor location (i.e., #locs = 1 and the system automated 6 edits (#suggestions = 6). Further, since 3 edits (e_5, e_6, and e_7) were automated using BluePencil$_{\text{auto}}$, we say that these locations are *automatically inferred* and write #inferredLocs = 3. We use the same experimental parameters as Section 6.3.

Evaluation Results. Table 4 shows the results of our simulation. In Setting 1, BluePencil required 191 examples and produced suggestions for 159 out of 350 edits *i.e.*, the synthesis engine assisted the developer to automate 45% of the edits. Meanwhile, in Setting 2, the synthesis engine automated

Table 4. Summary of the comparison to BluePencil. Column #inferredLocs is the number of additional inputs that are automatically identified by the feedback system. Column %automated shows the percentage of edits automated by the synthesis engine.

Approach	Edit	#examples	#locs	#inferredLocs	#suggestions	%automated	Time(s)
Setting 1	350	191	-	-	159	45%	0.25
Setting 2	350	87	87	37	263	75%	0.32

263 edits, which represents 75% of the total number of edits. It required only 87 examples and 87 cursor locations. Additionally, BluePencil$_{\text{auto}}$ found 37 additional inputs, decreasing the number of cursor locations the developer has to provide.

While Setting 2 (BluePencil$_{\text{cur}}$ and BluePencil$_{\text{auto}}$) is more effective at producing suggestions, the tool should also be fast enough to ensure that the suggestions can be generated at run-time when developers are programming. Therefore, we also evaluated its efficiency by measuring the time to generate edit suggestions. Column Time displays the averaged time to generate edit suggestions. Our engine produced suggestions in 0.32 seconds on average, fast enough to be used as an on-the-fly synthesizer in an IDE. Compared to BluePencil, it was slightly slower as it continuously refines the transformation by invoking the synthesis engine multiple times.

In Setting 2 (BluePencil$_{\text{cur}}$ and BluePencil$_{\text{auto}}$), the synthesis engine automated 75% of the edits, compared to 45% edits automated in Setting 1 (BluePencil). On average, our engine took 0.32 seconds to produce suggestions.

6.6 Discussion

In the above experiments, our technique produced a small number of false positives and false negatives. Besides false positives and negatives related to the limitations of Refazer itself, we found false positives related to semi-supervised synthesis and the automated feedback oracles. We also observed false positives related to the limitations of our anti-unification algorithm.

The semi-supervised synthesis technique produces a false positive in the following case. Given the edit: `Model(. . . , outputs: null, inputs: null)` ↦ `Model(. . . , outputs: null)`, i.e., removing `inputs: null`, and the additional positive input: `Model(. . . , inputs: new List<ModelInput>(), outputs: null)`, semi-supervised synthesis generates a transformation that deletes the last argument. (Note that the order of the last two parameters has been reversed.) Therefore, the synthesized transformation will produce the suggestion for the additional input by deleting the last argument `outputs: null`. However, the desired edit is deleting the `inputs` : ∗ clause, which is the second last argument in the additional input. That is, the correct suggestion should be to remove the second-to-last argument.

One way to address this issue would be to extend the anti-unification algorithm to handle commutativity as the order of "`name: value`" style arguments is irrelevant. However, this would complicate our anti-unification problem, with having to handle standard arguments under the AU (associativity and unity) theory and the named arguments under the ACU (associativity, commutativity, and unity) theory.

Limitations of the Feedback Oracles. In our experiment, BluePencil$_{\text{cur}}$ and BluePencil$_{\text{auto}}$ produced false positives and negatives due to limitations in the feedback oracles. It might classify negative inputs as positive ones if the locations are too similar. For instance, developers made the following edit: `comparedEdge.Item2 >= Source.Index` ↦ `comparedEdge.Item2 > Source.Index`.

The developer's intention was to change `>=` to `>` only if the left side of the comparison expression was `comparedEdge.Item2`. The oracle classified `comparedEdge.Item1 >= Source.Index` as a positive addition since the input is very similar. As future work, we plan to allow users to provide feedback about false positives, so that the system can create negative inputs. On the other hand, the false negatives mainly happened on small inputs where the change was on the root of the AST. In this case, any generalization of the input looked like an over generalization for the feedback oracle, since there was not much context for transformation.

Threats to Validity. Our benchmark suite may not be representative of the different types of edits developers perform. To reduce this threat, we collected real-world scenarios from developers who are working on different large code-bases to have as much variety as possible in the benchmark suite. Another threat is that developers may perform irrelevant, non-repetitive edits in addition to the repetitive ones, which may affect the effectiveness of our technique. To alleviate this issue, we also collected the traces of irrelevant edits and used them in our benchmarks. Finally, in some scenarios of repetitive edits, it is difficult even for humans to discern the transformation intended by the developer, which may affect the construction of our benchmark. To reduce this threat, multiple authors of the paper reviewed these ambiguous scenarios. Wherever possible, we contacted the developer who made the edit for confirmation.

7 RELATED WORK

Program Synthesis. Program synthesis, while being an old field of study [Buchi and Landweber 1969; Manna and Waldinger 1980; Pnueli and Rosner 1989], has recently been successfully used in many domains including data manipulation and wrangling [Gulwani 2011; Singh 2016; Yaghmazadeh et al. 2018], data structure manipulation and design [Feser et al. 2015; Frankle et al. 2016; Singh and Solar-Lezama 2011], concurrent programming [Cerný et al. 2011, 2013; Solar-Lezama et al. 2008; Vechev et al. 2010], and distributed controller design [Alur et al. 2014; Udupa et al. 2013]. The counter-example guided inductive synthesis procedure, that turns any synthesis task into repeated solving of programming-by-example tasks is the basis of the state-of-the-art synthesis technique Sketch [Solar-Lezama et al. 2005; Solar-Lezama et al. 2006; Solar-Lezama et al. 2006, 2007]. The syntax guided synthesis (SyGuS) framework [Alur et al. 2013] attempts to unify synthesis tasks from different domains by providing a mechanism to specify both the syntax and semantics of the desired solution. Efficient general purpose SyGuS solvers have been built and have found success in various domains [Alur et al. 2015, 2017; Huang et al. 2020; Reynolds et al. 2015; Udupa et al. 2013]. However, general purpose synthesizers are often less efficient than domain-specific ones as they are not able to leverage domain specific algorithms and techniques. Further, in most program synthesis techniques, the specification needs to be well-defined and provided explicitly. In our setting, no explicit specification is available: our technique automatically determines which subset of edits from a history of edits should be the example specification for the synthesis task. This ability to automatically determine which examples to use allows for the modeless operation of our technique and its predecessor BluePencil.

Interactive Program Synthesis. Interactive program synthesis systems allow users to incrementally refine the specification in response to synthesizer outputs [An et al. 2019; Le et al. 2017]. Within this paradigm, a notable approach for proposing refinements is based on the concept of distinguishing inputs [Jha et al. 2010], in which inputs are discovered for which the outputs of multiple consistent programs disagree, suggesting the need for additional refinement to rule out undesired candidate programs. FlashProg [Mayer et al. 2015] employs this notion of distinguishing inputs to pose parsimonious sequences of questions to the user to resolve ambiguities with respect to the user's specification. A disadvantage of this approach, however, is the overhead required for users to answer potentially many rounds of clarifying questions to refine intent. In this paper, we propose a complementary technique: we can synthesize new programs using semi-supervised synthesis. Our approach has the advantage that it allows users to refine intent with little to no modification to their workflow. Additionally, the technique not only leverages user feedback but also allows fully automated feedback during specification refinement.

Semi-Supervised Learning. Semi-supervised machine learning techniques [Zhu and Goldberg 2009] combine labeled data (i.e., input-output examples) with unlabeled data (i.e., additional inputs) during training to exploit a large amount of unlabeled data available in many domains, such as websites, source code, and images [Zhu 2005]. Beyond classical machine learning settings, semi-supervised learning techniques have also been adapted for use in program synthesis. For example, the BlinkFill system [Singh 2016] for synthesizing spreadsheet string transformations exploits input data by extracting a graphical constraint system to efficiently encode the logical structure across all available inputs. This input structure allows BlinkFill to achieve dramatic reduction in the number of candidate programs, leading to improvement in performance and reduction in the number of input-output examples over previous systems [Gulwani 2011]. Unfortunately, direct application of this approach to the domain of program transformations is impractical due to different types of inputs (positive inputs and negative inputs), the large number of inputs (all AST nodes in the source code) and the size of the ASTs themselves (potentially many thousands of tokens per file). To mitigate these issues, we have proposed a novel technique based on reward functions to isolate only those additional inputs that are likely to provide fruitful disambiguation, while still preserving the runtime efficiency required for interactive use in an IDE setting.

Software Refactoring Tools. Software refactorings are structured changes to existing software that improve code quality while preserving program semantics [Mens and Tourwe 2004; Opdyke 1992]. Popular IDEs such as Visual Studio [Microsoft 2019], Eclipse [Eclipse Foundation 2020], and IntelliJ [JetBrains 2020a] provide built-in support for various forms of well-understood software refactorings. However, experience shows that these refactoring tools are often underutilized by developers [Vakilian et al. 2012]. Impediments to adoption include the tedium associated with applying refactorings, and lack of awareness that a desired refactoring exists (the discoverability problem). Additionally, recent studies [Kim et al. 2012] indicate that developers often relax the requirement on semantics-preservation in practice, suggesting the need for tools for ad-hoc repetitive code transformation [Steimann and von Pilgrim 2012], not just well-known refactorings.

Several program synthesis-based approaches have been studied toward user-friendly refactoring and code transformation support, such as the Sydit, Lase, and ReFazer systems [Meng et al. 2011, 2013; Rolim et al. 2017] for synthesis of code transformations from examples. Getafix [Bader et al. 2019] and Revisar [Rolim et al. 2018] apply code mining techniques to discover such changes offline from large codebases, thus expanding breadth while also mitigating the burden for users to specify examples explicitly. BluePencil [Miltner et al. 2019] takes an alternative approach to increase discoverability and user-friendliness: the system uses a modeless, on-the-fly interaction model in which the programmer is presented with suggested edits without ever exiting the boundaries of the IDE's text editor–the system watches the user's behavior and analyzes code change patterns to discover ad-hoc repetitive edits.

The semi-supervised feedback learning approach in this paper is complementary and compatible with the techniques employed by BluePencil: the modeless interaction of BluePencil provides easy discoverability, and additional inputs provide a natural and effective mechanism for refinement when a false negative or positive is discovered.

Code Suggestions. Related to refactoring by example are techniques for suggesting code completions. [Raychev et al. 2014] train statistical language models to predict API usage patterns from code snippets extracted from websites such as Github and StackOverflow. These models are capable of filling partial programs with holes corresponding to missing method names or parameters. The Bing Developer Assistant [Zhang et al. 2016] also employs statistical models for code snippets, but for the purpose of answering natural language code search queries. MatchMaker [Yessenov et al. 2011] analyzes dynamic executions, rather than source code, of real-world programs for API usage patterns. The aforementioned approaches all require training over large datasets, whereas our approach provides suggestions from few examples and additional inputs. In contrast to statistical techniques, type-based code completion approaches exploit type information to complete partial expressions [Gvero et al. 2013; Perelman et al. 2012]. Because these techniques require rich type information, they may be difficult or impractical to apply toward dynamically-typed languages. Our approach avoids this difficulty by requiring only syntax trees.

8 CONCLUSION AND FUTURE WORK

Developer tools that proactively predict users' actions and help them improve their productivity are gaining popularity. In this context, we presented a novel approach for predicting repeated edits that exploits the latent information in the user's code. By combining knowledge about what edits the user has performed in the past with the observable patterns in rest of the code, our technique is able to significantly improve precision and recall metrics for predicting future repeated edits. It is intriguing to think about the potential of harnessing other forms of hidden information in user's code and actions to ease the task of producing bug-free code revisions.

We are integrating the proposed techniques in Visual Studio IntelliCode suggestions. A screen capture of the current prototype is presented in Figure 2. After the integration, we plan to conduct user studies to evaluate the usability of the system. Additionally, we plan to develop different user interface designs to allow developers to provide additional inputs in different ways. Another line of work we are excited to explore is the use of large scale edit history data to produce suggestions not only for a sequence of repetitive edits, but any sequence of edits predictable from past data. We also plan to explore the use of our technique in other programming-by-example domains such as text manipulation and relational data wrangling. Another exciting future direction is to use BluePencil$_{\text{auto}}$ to produce an interactive code review assistant. In summary, we believe that the techniques presented in this paper can enable many code editing and reviewing tools for many application scenarios with varying levels of user interaction.

ACKNOWLEDGMENTS

We thank Peter Groenewegen, David Obando, Keith Simmons, and Mark Wilson-Thomas for providing valuable feedback and support during this study. Additionally, we thank the Microsoft Visual Studio IntelliCode team and other Microsoft developers who provided data and helped us in validating the design of BluePencil$_{\text{cur}}$. We also thank the anonymous reviewers for comments and suggestions that helped us greatly improve the paper.

REFERENCES

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 *2013 Formal Methods in Computer-Aided Design*, pages 1–8, 2013.

R. Alur, M. M. K. Martin, M. Raghothaman, C. Stergiou, S. Tripakis, and A. Udupa. Synthesizing finite-state protocols from scenarios and requirements. In E. Yahav, editor, *10th International Haifa Verification Conference*, volume 8855 of *Lecture Notes in Computer Science*, pages 75–91. Springer, 2014. doi: 10.1007/978-3-319-13338-6_7. URL https://doi.org/10.1007/978-3-319-13338-6_7.

R. Alur, P. Černỳ, and A. Radhakrishna. Synthesis through unification. In *International Conference on Computer Aided Verification*, pages 163–179. Springer, 2015.

R. Alur, A. Radhakrishna, and A. Udupa. Scaling enumerative program synthesis via divide and conquer. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 319–336. Springer, 2017.

S. An, R. Singh, S. Misailovic, and R. Samanta. Augmented example-based synthesis using relational perturbation properties. *Proceedings of the ACM on Programming Languages*, 4(POPL):1–24, 2019.

J. Bader, A. Scott, M. Pradel, and S. Chandra. Getafix: Learning to fix bugs automatically. *Proc. ACM Program. Lang.*, 3 (OOPSLA), Oct. 2019. doi: 10.1145/3360585. URL https://doi.org/10.1145/3360585.

A. Bessey, K. Block, B. Chelf, A. Chou, B. Fulton, S. Hallem, C. Henri-Gros, A. Kamsky, S. McPeak, and D. Engler. A few billion lines of code later. *Commun. ACM*, 53(2), 2010. ISSN 0001-0782. doi: 10.1145/1646353.1646374. URL https://doi.org/10.1145/1646353.1646374.

J. R. Buchi and L. H. Landweber. Solving sequential conditions by finite-state strategies. *Transactions of the American Mathematical Society*, 138:295–311, 1969. ISSN 00029947. URL http://www.jstor.org/stable/1994916.

P. Cerný, K. Chatterjee, T. A. Henzinger, A. Radhakrishna, and R. Singh. Quantitative synthesis for concurrent programs. In G. Gopalakrishnan and S. Qadeer, editors, *23rd International Conference on Computer Aided Verification (CAV)*, volume 6806 of *Lecture Notes in Computer Science*, pages 243–259. Springer, 2011. doi: 10.1007/978-3-642-22110-1_20. URL https://doi.org/10.1007/978-3-642-22110-1_20.

P. Cerný, T. A. Henzinger, A. Radhakrishna, L. Ryzhyk, and T. Tarrach. Efficient synthesis for concurrency by semantics-preserving transformations. In N. Sharygina and H. Veith, editors, *25th International Conference on Computer Aided Verification (CAV)*, volume 8044 of *Lecture Notes in Computer Science*, pages 951–967. Springer, 2013. doi: 10.1007/978-3-642-39799-8_68. URL https://doi.org/10.1007/978-3-642-39799-8_68.

Eclipse Foundation. Eclipse. At https://www.eclipse.org/, 2020.

W. S. Evans, C. W. Fraser, and F. Ma. Clone detection via structural abstraction. *Software Quality Journal*, 17(4):309–330, 2009.

J. K. Feser, S. Chaudhuri, and I. Dillig. Synthesizing data structure transformations from input-output examples. In D. Grove and S. Blackburn, editors, *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*, pages 229–239. ACM, 2015. doi: 10.1145/2737924.2737977. URL https://doi.org/10.1145/2737924.2737977.

J. Frankle, P. Osera, D. Walker, and S. Zdancewic. Example-directed synthesis: a type-theoretic interpretation. In R. Bodík and R. Majumdar, editors, *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, pages 802–815. ACM, 2016. doi: 10.1145/2837614.2837629. URL https://doi.org/10.1145/2837614.2837629.

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 (POPL '11)*. ACM New York, NY, USA, 2011.

T. Gvero, V. Kuncak, I. Kuraj, and R. Piskac. Complete completion using types and weights. In *Proceedings of the 34th ACM SIGPLAN conference on Programming language design and implementation*, pages 27–38, 2013.

K. Huang, X. Qiu, P. Shen, and Y. Wang. Reconciling enumerative and deductive program synthesis. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 1159–1174, 2020.

JetBrains. IntelliJ. At https://www.jetbrains.com/idea/, 2020a.

JetBrains. ReSharper. At https://www.jetbrains.com/resharper/, 2020b.

S. Jha, S. Gulwani, S. A. Seshia, and A. Tiwari. Oracle-guided component-based program synthesis. In *2010 ACM/IEEE 32nd International Conference on Software Engineering*, volume 1, pages 215–224. IEEE, 2010.

L. Jiang, G. Misherghi, Z. Su, and S. Glondu. Deckard: Scalable and accurate tree-based detection of code clones. In *29th International Conference on Software Engineering (ICSE'07)*, pages 96–105. IEEE, 2007.

M. Kim, T. Zimmermann, and N. Nagappan. A field study of refactoring challenges and benefits. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, FSE '12, pages 50:1–50:11, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1614-9. doi: 10.1145/2393596.2393655. URL http://doi.acm.org/10.1145/2393596.2393655.

V. Le, D. Perelman, O. Polozov, M. Raza, A. Udupa, and S. Gulwani. Interactive program synthesis. *arXiv preprint arXiv:1703.03539*, 2017.

Z. Manna and R. J. Waldinger. A deductive approach to program synthesis. *ACM Transactions on Programming Languages and Systems*, 2(1):90–121, 1980. doi: 10.1145/357084.357090. URL https://doi.org/10.1145/357084.357090.

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 & Technology*, pages 291–301, 2015.

N. Meng, M. Kim, and K. S. McKinley. Systematic editing: generating program transformations from an example. *ACM SIGPLAN Notices*, 46(6):329–342, 2011.

N. Meng, M. Kim, and K. S. McKinley. Lase: locating and applying systematic edits by learning from examples. In *2013 35th International Conference on Software Engineering (ICSE)*, pages 502–511. IEEE, 2013.

T. Mens and T. Tourwe. A survey of software refactoring. *IEEE Transactions on Software Engineering*, 30(2):126–139, Feb 2004. ISSN 0098-5589. doi: 10.1109/TSE.2004.1265817.

Microsoft. Visual Studio. At https://www.visualstudio.com, 2019.

Microsoft. Intellicode suggestions. At https://docs.microsoft.com/en-us/visualstudio/intellicode/intellicode-suggestions, 2020.

A. Miltner, S. Gulwani, V. Le, A. Leung, A. Radhakrishna, G. Soares, A. Tiwari, and A. Udupa. On the fly synthesis of edit suggestions. *Proceedings of the ACM on Programming Languages*, 3(OOPSLA):1–29, 2019.

T. M. Mitchell. Generalization as search. *Artificial intelligence*, 18(2):203–226, 1982.

H. A. Nguyen, A. T. Nguyen, T. T. Nguyen, T. N. Nguyen, and H. Rajan. A study of repetitiveness of code changes in software evolution. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 180–190. IEEE, 2013.

W. F. Opdyke. *Refactoring Object-oriented Frameworks*. PhD thesis, Champaign, IL, USA, 1992. UMI Order No. GAX93-05645.

D. Perelman, S. Gulwani, T. Ball, and D. Grossman. Type-directed completion of partial expressions. In *Proceedings of the 33rd ACM SIGPLAN conference on Programming Language Design and Implementation*, pages 275–286, 2012.

A. Pnueli and R. Rosner. On the synthesis of a reactive module. In *Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 179–190, 1989.

O. Polozov and S. Gulwani. Flashmeta: a framework for inductive program synthesis. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 107–126, 2015.

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*, pages 419–428, 2014.

A. Reynolds, M. Deters, V. Kuncak, C. Tinelli, and C. Barrett. Counterexample-guided quantifier instantiation for synthesis in smt. In *International Conference on Computer Aided Verification*, pages 198–216. Springer, 2015.

R. Rolim, G. Soares, L. D'Antoni, O. Polozov, S. Gulwani, R. Gheyi, R. Suzuki, and B. Hartmann. Learning syntactic program transformations from examples. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, pages 404–415. IEEE, 2017.

R. Rolim, G. Soares, R. Gheyi, T. Barik, and L. D'Antoni. Learning quick fixes from code repositories, 2018.

R. Singh. Blinkfill: Semi-supervised programming by example for syntactic string transformations. *Proc. VLDB Endow.*, 9 (10):816–827, June 2016. ISSN 2150-8097. doi: 10.14778/2977797.2977807. URL https://doi.org/10.14778/2977797.2977807.

R. Singh and A. Solar-Lezama. Synthesizing data structure manipulations from storyboards. In T. Gyimóthy and A. Zeller, editors, *SIGSOFT/FSE'11 19th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-19) and ESEC'11: 13th European Software Engineering Conference (ESEC-13), Szeged, Hungary, September 5-9, 2011*, pages 289–299. ACM, 2011. doi: 10.1145/2025113.2025153. URL https://doi.org/10.1145/2025113.2025153.

A. Solar-Lezama, R. M. Rabbah, R. Bodík, and K. Ebcioglu. Programming by sketching for bit-streaming programs. In V. Sarkar and M. W. Hall, editors, *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation, Chicago, IL, USA, June 12-15, 2005*, pages 281–294. ACM, 2005. doi: 10.1145/1065010.1065045. URL https://doi.org/10.1145/1065010.1065045.

A. Solar-Lezama, L. Tancau, R. Bodik, S. Seshia, and V. Saraswat. Combinatorial sketching for finite programs. In *Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, pages 404–415, 2006.

A. Solar-Lezama, L. Tancau, R. Bodík, S. A. Seshia, and V. A. Saraswat. Combinatorial sketching for finite programs. In J. P. Shen and M. Martonosi, editors, *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2006, San Jose, CA, USA, October 21-25, 2006*, pages 404–415. ACM, 2006. doi: 10.1145/1168857.1168907. URL https://doi.org/10.1145/1168857.1168907.

A. Solar-Lezama, G. Arnold, L. Tancau, R. Bodík, V. A. Saraswat, and S. A. Seshia. Sketching stencils. In J. Ferrante and K. S. McKinley, editors, *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation, San Diego, California, USA, June 10-13, 2007*, pages 167–178. ACM, 2007. doi: 10.1145/1250734.1250754. URL https://doi.org/10.1145/1250734.1250754.

A. Solar-Lezama, C. G. Jones, and R. Bodík. Sketching concurrent data structures. In R. Gupta and S. P. Amarasinghe, editors, *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation, Tucson, AZ, USA, June 7-13, 2008*, pages 136–148. ACM, 2008. doi: 10.1145/1375581.1375599. URL https://doi.org/10.1145/1375581.1375599.

F. Steimann and J. von Pilgrim. Refactorings without names. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, ASE 2012, pages 290–293, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1204-2. doi: 10.1145/2351676.2351726. URL http://doi.acm.org/10.1145/2351676.2351726.

A. Udupa, A. Raghavan, J. V. Deshmukh, S. Mador-Haim, M. M. Martin, and R. Alur. Transit: specifying protocols with concolic snippets. *ACM SIGPLAN Notices*, 48(6):287–296, 2013.

M. Vakilian, N. Chen, S. Negara, B. A. Rajkumar, B. P. Bailey, and R. E. Johnson. Use, disuse, and misuse of automated refactorings. In *2012 34th International Conference on Software Engineering (ICSE)*, pages 233–243. IEEE, 2012.

M. T. Vechev, E. Yahav, and G. Yorsh. Abstraction-guided synthesis of synchronization. In M. V. Hermenegildo and J. Palsberg, editors, *Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2010, Madrid, Spain, January 17-23, 2010*, pages 327–338. ACM, 2010. doi: 10.1145/1706299.1706338. URL https://doi.org/10.1145/1706299.1706338.

N. Yaghmazadeh, X. Wang, and I. Dillig. Automated migration of hierarchical data to relational tables using programming-by-example. *Proc. VLDB Endow.*, 11(5):580–593, 2018. doi: 10.1145/3187009.3177735. URL http://www.vldb.org/pvldb/vol11/p580-yaghmazadeh.pdf.

K. Yessenov, Z. Xu, and A. Solar-Lezama. Data-driven synthesis for object-oriented frameworks. *ACM SIGPLAN Notices*, 46 (10):65–82, 2011.

H. Zhang, A. Jain, G. Khandelwal, C. Kaushik, S. Ge, and W. Hu. Bing developer assistant: improving developer productivity by recommending sample code. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 956–961, 2016.

X. Zhu and A. B. Goldberg. Introduction to semi-supervised learning. *Synthesis lectures on artificial intelligence and machine learning*, 3(1):1–130, 2009.

X. J. Zhu. Semi-supervised learning literature survey. Technical report, University of Wisconsin-Madison Department of Computer Sciences, 2005.