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

It is fairly common for developers to make repeated edits in code that are all instances of a more-general program transformation. Since this process can be tedious and error-prone, we study the problem automatically learning program transformations from past edits, which can then be used to predict future edits. We take the 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 fewer past edit data.

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 input-output 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 to production at Facebook where it is being used during the maintenance of Facebook apps. BluePencil (Miltner et al., 2019) produces code edit suggestions to automate repetitive code edits

*Xiang Gao performed this work as part of his internship with the Prose team at Microsoft.
†Shraddha Barke performed this work as part of his internship 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, gaoxiang@comp.nus.edu.sg; 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.

by synthesizing transformations on-the-fly based on the recent edits performed by the developer. It has been deployed in Microsoft Visual Studio (VS) and is available as part of VS IntelliCode.

The main challenge of generalizing examples of edits to program transformations lies in synthesizing an intended generalization that not only satisfies the few examples but also produces the correct edits on unseen inputs. Incorrect generalizations can lead to *false negatives*: the transformation does not produce an intended edit 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 lead 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) (Polozov and Gulwani, 2015b). It then uses a ranking system to rank the transformations and select the one that is more likely to be correct based on a set of predefined heuristics. However, despite the more sophisticated approach to generalization, Refazer still required up to six examples during its evaluation (Rolim et al., 2017).

All aforementioned techniques rely only on input-output examples and background knowledge (ranking functions) to deal with the generalization problem. Another alternative is to use semi-supervised learning (Zhu and Goldberg, 2009); an approach to machine learning that combines a set of input-output examples, and an amount of 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 source code can also provide many *additional inputs* where the synthesized transformation should and should not produce edits 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 < threshold)` 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) < threshold)`, 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 transformations synthesized using the new input through a user-driven or automated feedback system.

In this paper, we propose Sparse, a feedback-driven semi-supervised technique to synthesize program transformations. Its approach is based on our two key observations above. Initially, Sparse synthesizes a program transformation from input-output examples using ReFazer (Rolim et al., 2017). For the input-output example, Sparse tracks which sub-trees 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* for reasons that will be apparent later on. As an example, consider again the change `if (score < threshold)` to `if (IsValid(score))`. The expression `score` was used in the output—it is a selected node. Next, Sparse iterates over candidate additional inputs to find more examples to refine the generalization. For each candidate input, it performs two main steps:

- Sparse computes the *anti-unification* of the input examples and the candidate additional input. Anti-unification is a generalization process which can identify corresponding sub-trees among different input ASTs. For instance, it can identify that `score` corresponds to the `GetScore(run)` in the example input and candidate additional input `if (GetScore(run) < threshold)`. 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 < threshold` and `UnrelatedCondition()`, and the sub-tree `score` itself has no corresponding sub-tree, 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 then re-synthesize the parts of the transformations using also the new example. In our running scenario, the new example is `if (GetScore(run) < threshold)` ↦ `if (IsValid(GetScore(run)))`.
- Sparse 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. For instance, it can take into consideration user-provided feedback, such as whether the transformation should use a particular input. Such a feedback is usually easier to be provided by the user than another input-output example. However, the feedback can also use automated components that rewards, for example, additional inputs that are similar to the example inputs but generalize the transformation without overgeneralizing it. If the final reward score is above a certain threshold, Sparse accepts the additional input and synthesizes a new program transformation using the new example.

We implemented Sparse for the domain of C# program transformations. It uses the implementation of Refazer available in the PROSE SDK[1]. Further, we augmented the BluePencil (Miltner et al., 2019) algorithm with Sparse 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:

- Sparse $_{\text{loc}}$: *User-provided feedback about missed 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 an input-output example. On a benchmark of 12,642 test cases, we compared Sparse $_{\text{loc}}$ with the baseline (ReFazer). While the recall of Blue-Pencil ranged from 26.71% (1 example provided) to 89.10% (3 examples provided), Sparse $_{\text{loc}}$'s recall was at least 99.94% and its precision was at least 96.01% with just 1 example (and 1 additional input) provided. These results

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

suggest that Sparse $_{loc}$ can synthesize suggestions with high precision to locations indicated by developers as false negatives.

- Sparse $_{cur}$: *Semi-automated feedback based on cursor position.* This feature uses the cursor position in the editor to indicate candidate additional inputs to Sparse. This feature is motivated by the fact 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 like a proxy for the user, and indicates implicitly, that the user wants to modify the current location. However, the cursor location is ambiguous. The sub-tree that the user wants to edit may be any of sub-trees that are present at the cursor location, i.e., lowest leaf node at the cursor location all the way to the root of the AST. The tool relies on feedback from Sparse's reward function (Section 4.2) to accept additional inputs. We compared this reward function with two alternative reward functions: (i) *no validation*, i.e.,, Sparse accepts any additional inputs; (ii) and *clone detection*, which accepts inputs based on their similarities with the input examples. Our results show that while "no validation" and "clone detection" lead to high false positives and negatives, respectively, Sparse's reward function produces only 11 false positives and 14 false negatives on 243,682 tested additional inputs. We also evaluated the effectiveness of Sparse $_{cur}$ in generating correct suggestions at the cursor location. Within 291 scenarios, Sparse $_{cur}$ only generates one false positive and three false negatives.
- Sparse $_{auto}$: *Automated feedback based on all inputs in the source code.* This feature uses all the nodes available in the source code as input to Sparse. This feature is relevant in the settings where user feedback is not feasible. 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 Sparse $_{auto}$ can save developers from indicating the additional inputs. In 86 scenarios of repetitive tests with a total of 350 repetitive edits, Sparse $_{loc}$ decreased the number of times the developer would have to indicate the input by 30%. When compared to Blue-Pencil, our results show that Sparse $_{auto}$ automated 263 edits while Blue-Pencil automated only 150.

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

- We formalize the feedback-driven semi-supervised synthesis problem (Section 3);
- We propose Sparse, the first known technique to address the feedback-driven semi-supervised synthesis problem for program transformations (Section 4);
- We propose three practical applications of Sparse and instantiate them for the domain of C# program transformations (Section 5);
- We evaluate Sparse along the dual axes of effectiveness (quality as measured by false positive and negative rates) and efficiency (user burden as measured by number of examples and additional inputs). Our results show that Sparse achieves near 100% precision and recall across almost 86 real-world developer scenarios, all while delivering each suggestion in less than half a second.

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 callsites where the developer has manually updated the invocation to match this

[2]Nuget is a the 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 to 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). Notice 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).

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, neither Visual Studio (Microsoft, 2019) nor ReSharper (JetBrains, 2020b) include built-in transformations to automate these edits.

However, a recently introduced Visual Studio feature based on BluePencil (Miltner et al., 2019), called IntelliCode Refactorings (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

Fig. 2. SPARSE $_{cur}$ 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. SPARSE $_{cur}$ 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.

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 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 SPARSE can help alleviating this problem. SPARSE $_{cur}$ 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, SPARSE $_{cur}$ 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). We provide details of our technique and its applications in Sections 4 and 5, respectively. 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 of program transformation problem and follow-up with 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 sub-trees in t. Each node in the AST consists of a string label representing the node type (e.g., Identifier, InvokeExpression, etc), set of attributes (e.g., text value of leaf nodes, etc) and a list of children.

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 report, we assume that each edit program P is a pair $(\mathsf{P}_{\mathsf{guard}}, \mathsf{P}_{\mathsf{trans}})$ of two parts: (a) a *guard* $\mathsf{P}_{\mathsf{guard}} : \mathbb{T} \to \mathbb{B}$, 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.

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

ReFazer learns this program initially in Section 2 (with just 2 examples). This program is written in terms of templates with each 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 input $\mathsf{PI} = \{\mathsf{pi}_0, \ldots, \mathsf{pi}_n$, and (c) a set of additional negative inputs $\mathsf{NI} = \{\mathsf{ni}_0, \ldots, \mathsf{ni}_m\}$, the *semi-supervised programming-by-example 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) $\forall 0 \leq j \leq n.\mathsf{P}(\mathsf{pi}) \neq \bot$, 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 and use that feedback to discover these additional inputs. In this setting, the synthesizer is provided some extra inputs: (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 sub-trees of the AST representing a source file. (b) A *reward function* $\mathsf{Rew} : \mathsf{InputPool} \nrightarrow [-\infty, \infty]$. The reward function acts as a feedback mechanism. A positive and negative reward for an $\mathsf{i} \in \mathsf{InputPool}$ indicates whether the synthesized program is applicable or not on i. For convenience we separate the rewards function in to the *user provided* $\mathsf{Rew}_{\mathsf{U}}$ and *environment provided* $\mathsf{Rew}_{\mathsf{E}}$ reward functions, with $\mathsf{Rew} = \mathsf{Rew}_{\mathsf{U}} + \mathsf{Rew}_{\mathsf{E}}$. In Section 4.2,

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

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 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/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 of 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 modelled 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 for the synthesized program is *how well does the synthesized program match user intent?* This criterion is hard to capture in any formally 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 semantics 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-dependant feedback stop changing (i.e., Rew_U is fixed), the synthesized program should converge to a fixed one. Due to the lack of strict correctness conditions, to ensure the quality of the programs and edits produced, we further experimentally validate the techniques with a comprehensive evaluation (see Section 6).

3.2 Background: Program-by-Example for Code

Programming-by-Example techniques form the basis of the techniques used in our proposed solution. For a given input domain $\mathbb{I}$, output domain $\mathbb{O}$, and class of programs 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 $\mathsf{P} : \mathbb{I} \to \mathbb{O}$ 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 technique. In ReFazer, the programs are drawn from the domain-specific language (DSL) shown in Figure 3. The programs are composed of guard and transformers as stated above. 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 selection returns a subtree of the input. The subtree is identified as the n^{th} node that satisfies a guard.

```
program := (guard, transformer)
guard   := pred | Conjunction(pred, guard)
pred    := IsNthChild(node, n)
         | IsKind(node, kind)
         | Attribute(node, attr) = value
         | Not(pred)

node    := Path(input, path)
```

```
transformer := construct | select
   construct      := Tree(kind, attrs, children)
   children       := EmptyChildren | Cons(node, children)
      | InsertChild(Children(select), pos, node)
      | DeleteChild(Children(select), pos)
      | ReplaceChildren(Children(select), posList, children)
      | MapChildren(λ input: transformer, Children(select))
   select         := Nth(Filter(guard, SubTrees(input)), n)
   pos            := n | ChildIndexOf(node)
```

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

Fig. 3. Domain-Specific language for edit programs

- Constructions: A construction return a subtree that is constructed 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 a additional node at the position `pos`.

We do not provide any 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, 2015a, Rolim et al., 2017) for further details.

Example 3.2. Revisit the edits in Figure 1a and Example 3.1, ReFazer synthesizes the following transformer: $X_1.X_2(X_3, X_4, X_5, X_6, X_7) \nrightarrow \text{DependencyResolveUtility}.X_2(X_1, X_3, X_4, X_5, X_6, X_7)$. The ReFazer program that represents this transformer is given by

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

Here, `select1`, `select2`, and `select3` extract the fragments X_2, X_1, and X_3, X_4, X_5, X_6, X_7. Each select is specified by guard. For example, for `select1`, the guard might be of the form `IsKind(Current, Identifier)` $\wedge$ `Attribute(Current, TextValue)` = `Resolve`$\wedge$ `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, and love to hate the tool (Bessey et al., 2010). Hence, most synthesis techniques, including ReFazer, used in the source code transformation domain err on the side of under-generalization (see, for example, (Bader et al., 2019, Meng et al., 2013, Rolim et al., 2017)).

4 FEEDBACK-DRIVEN SEMI-SUPERVISED SYNTHESIS

We present the Sparse 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 n and p with $n < p$, with $\text{PI} = \{\text{i} \in \text{InputPool} \mid \text{Rew}_{n-1}(\text{i}) > p\}$ and $\text{NI} = \{\text{i} \in$

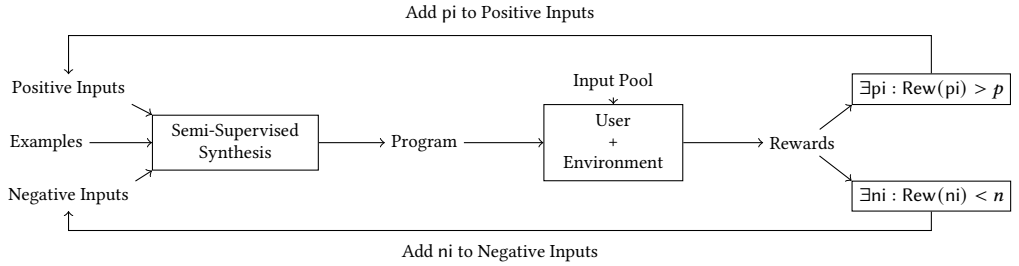

Fig. 4. Solution approach for the feedback-driven semi-supervised problem

InputPool | $\mathsf{Rew}_{n-1}(\mathsf{i}) < n\}$. The thresholds p and n may depend on a wide variety of factors including the number of examples in $\mathsf{Examples}_n$, properties of the program P_{n-1}, etc. See Section 4.2.

• The semi-supervised synthesis is solved using a standard (not semi-supervised) programs 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 by using a given example $\mathsf{i} \mapsto \mathsf{o}$, and using a combination of provenance analysis and anti-unification. Informally, we first associate each sub-tree s' of pi with an equivalent sub-tree s of i. Then, in o we replace each sub-tree generated from a sub-tree s of the input i, with a new sub-tree that is generated in a similar way but with s replaced by s'. See Section 4.1.

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 P

```
1:  Inputs ← {i | (i ↦ o) ∈ Examples}
2:  P_guard ← ReFazer_guard(Inputs ∪ PI, NI)
3:  P_trans ← TransSynth(Examples, PI)
4:  if P_guard = ⊥ ∨ P_trans = ⊥ then return ⊥
5:  return (P_guard, P_trans)
6:
7:  function TransSynth(Examples, PI)
8:      P_trans ← ReFazer_trans(Examples)
9:      π ← Provenance(i_0 ↦ o_0, P_trans)
10:     (τ, ⟨σ_0, ..., σ_k, σ'_0, ..., σ'_n⟩) ← ⋈_π{i_0, ..., i_k, pi_0, ..., pi_n}
11:     if ⊥ ∈ (σ_0, ..., σ_k, σ'_0, ..., σ'_n) then return ⊥
12:     AdditionalExamples ← {pi_j → Evaluate*(P_trans, pi, i) | pi_j ∈ PI}
13:     return ReFazer_trans(Examples ∪ AdditionalExamples)
```

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

The guard synthesis component of the algorithm (line 2) falls back to $\mathsf{ReFazer}_{\mathsf{guard}}$. However, transformer synthesis is significantly more involved. First, using only the examples 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 the *anti-unification* (line 10) to determine which fragments of the example inputs are associated 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.

Provenance. The first step of transformer synthesis computes *provenance* information for each example. The provenance information is computed for selects. Given a transformer program $\mathsf{P}_{\mathsf{trans}}$, 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 sub-tree of i, and (b) each SP_j is a sub-program of $\mathsf{P}_{\mathsf{trans}}$ that is a `select`, and SP_j produces the output si_j during the execution of $\mathsf{P}_{\mathsf{trans}}(\mathsf{si})$. Note that each SP_j may have multiple sub-trees si_j and si'_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. The need for provenance information will be clear in the rest of this section. One reason we consider provenance only for selects is that any `guard` operation is going to depend all of the input tree.

Example 4.1. Consider the $\mathsf{P}_{\mathsf{trans}}$ shown in Example 3.2 and with the abbreviated example:
`repository.ResolveDependency(dependency1, args...)` ↦
`DependencyResolverUtility.ResolveDependency(repository, dependency1, args...)`
The provenance information is given by π = { `select1` ← `ResolveDependency`, `select2` ← `repository`, `select3` ← `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 $(\tau, \langle \sigma_1, \sigma_2 \rangle)$ such that:

- a *template* τ, i.e., an AST with labelled holes $\{\mathsf{h}_0, \ldots, \mathsf{h}_n\}$, and
- two substitutions $\sigma_1, \sigma_2 : \{\mathsf{h}_0, \ldots, \mathsf{h}_n\} \rightarrow \mathbb{T}$ 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 $\bowtie\{\mathsf{i}_1, \ldots, \mathsf{i}_n\}$. As is standard, we write anti-unification to mean the anti-unification that produces the most specific generalization.

Example 4.2. Consider two inputs i_1 = `if(score < threshold)` and i_2 = `if (GetScore(run) < threshold)`. The anti-unification $\bowtie\{\mathsf{i}_1, \mathsf{i}_2\}$ = $\mathsf{if}(\mathsf{h}_0 - \mathsf{threshold}), \langle \{\mathsf{h}_0 \mapsto \mathsf{score}\}, \{\mathsf{h}_0 \mapsto \mathsf{GetScore(run)}\} \rangle$. 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 homogenous 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.

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

- If the roots of i_1 and i_2 have different labels or attributes: then $i_1 \bowtie i_2 = (h, (\{h \mapsto i_1\}, \{h \mapsto i_2\}))$.
- If the root nodes of i_1 and i_2 have the same label label and attributes attrs, and if the nodes fixed-arity: then $i_1 \bowtie i_2 = \text{Tree}(\text{label}, \text{attrs}, \tau_1 \ldots \tau_n), \langle \bigcup_i \sigma_2^i, \bigcup_i \sigma_2^i \rangle$ where (a) $\text{Children}(i_1) = i_1^1, \ldots, i_1^n$ and $\text{Children}(i_2) = i_2^1, \ldots, i_2^n$, and (b) for all $1 \leq i \leq n . i_1^j \bowtie i_2^j = (\tau_j, (\sigma_1^j, \sigma_2^j))$
- 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 $i_1^1, \ldots, i_1^n$ and $i_2^1, \ldots, i_2^m$, respectively. Then, we compute two lists of node sequences $s^0, d_i^0, s^1 \ldots d_i^k, s^k$ for $i \in \{1, 2\}$ such that: (a) The concatenation of $s^0 d_i^0 s^1 \ldots d_i^k s^k$ is equal to $i_1^1, \ldots, i_1^n$ and $i_2^1, \ldots, i_2^n$ for $i = 1$ and $i = 2$, respectively. Note that s^i are shared between the two lists. (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 $i_1 \bowtie i_2 = (\text{Tree}(\text{label}, \text{attrs}, s^1 h_1 \ldots s^k), \langle \{h_i \mapsto d_1^i \mid 0 \leq i \leq k\}, \{h_i \mapsto d_2^i \mid 0 \leq i \leq k\} \rangle)$.

Remark 4.3. The anti-unification of two ASTs i_1 and i_2 is not uniquely defined in some cases. For example, the both i_1 and i_2 be argument lists with $i_1 = (\texttt{var}, \texttt{var})$ and $i_1 = (\texttt{var})$. Now, $i_1 \bowtie i_2$ is computed as per the third case above. As per the definition, we have two options for the result: (a) $((\texttt{var}, h), \langle \{h \mapsto \texttt{var}\}, \{h \mapsto \texttt{nil}\} \rangle$, or (b) $((h, \texttt{var}), \langle \{h \mapsto \texttt{var}\}, \{h \mapsto \texttt{nil}\} \rangle$. That is, it is unclear if the var in i_2 matches with the first or the second var in i_1. This issue can be resolved by using more advanced anti-unification techniques. However, we did not notice such cases in our experiments.

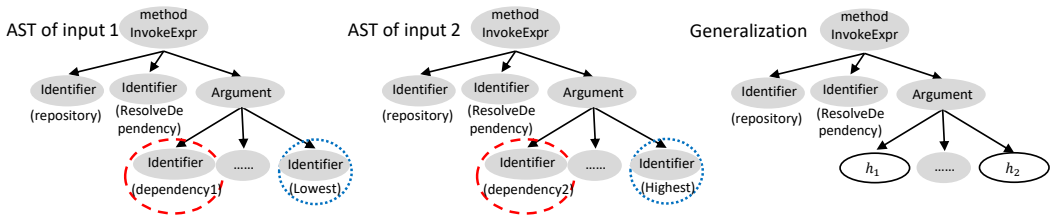

Fig. 5. The partial AST of two inputs shown in Figure 1a and 1c, and their anti-unification.

For our use case, we do not consider the general notion of anti-unification, but *anti-unification modulo provenance*. Informally, we need every substitution to be above all `select` operations. That is, if $h \mapsto si$, then $\nexists SP, si' : (SP \leftarrow si') \in \pi \wedge si'$ is an ancestor of si. Note that the additional constraint makes the anti-unification be undefined in certain cases (see Example 4.4). We use the notation $i_1 \bowtie_\pi i_2$ and $\bowtie_\pi \{i_1, \ldots, i_n\}$ to denote anti-unification modulo provenance.

Example 4.4. Consider the inputs i_1 = `if(GetScore(run) < threshold)` and i_2 = `if(score < threshold)`. Without provenance, anti-unification generates substitutions $\sigma_1 = \{h \mapsto \texttt{GetScore(run)}\}$ and $\sigma_2 = \{h \mapsto \texttt{score}\}$. However, if we have provenance information $\pi = \{\texttt{select1} \leftarrow \texttt{run}\}$, then no anti-unification exists modulo π. Intuitively, we are trying to match "important parts" (here, selected nodes) of i_1 with equivalent parts in i_2. However, there is no equivalent to run in i_2.

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

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

- For every sub-program SP of $\mathsf{P}_{\mathsf{trans}}$ which is not a `select` and has no sub-program that is a `select`, the evaluation value is set to SP(i).
- For every other sub-program, the evaluation value is computed in the standard way, i.e., using the semantics of the operator and the evaluation values of the child sub-programs.

We do not describe this procedure formally, but illustrate it using an example.

Example 4.5. Consider the first input in Figure 1a and 1c, anti-unification generates $\sigma_1 = \{\mathsf{h}_1 \mapsto \textit{dependency}, \mathsf{h}_2 \mapsto \textit{Lowest}\}$ and $\sigma_2 = \{\mathsf{h}_1 \mapsto \textit{dependency2}, \mathsf{h}_2 \mapsto \textit{Highest}\}$. To produce the output for the additional positive input in 1c, we apply $\sigma_2(\sigma_1^{-1}(\mathsf{si}))$ to every $\mathsf{SP} \leftarrow \mathsf{si} \in \pi$. Here, these sub-trees are: `dependency` and `Lowest`, and several others selects which are not ancestors of h_1 or h_2 (e.g., `ResolveDependency`). Now, we have $\sigma_2(\sigma_1^{-1}(\texttt{dependency})) =$ `dependency1` and $\sigma_2(\sigma_1^{-1}(\texttt{Lowest})) =$ `Highest`. 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 examples and the new examples generated from additional positive inputs to the transformer synthesis component of the ReFazer. We have the following theorem.

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

The proof follows from the use of $\mathsf{ReFazer}_{\mathsf{trans}}$ and $\mathsf{ReFazer}_{\mathsf{guard}}$ in lines 13 and 2, respectively.

Remark 4.7 (Completness 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 $\mathsf{Examples} = \{\text{“}\texttt{(temp - 32) * (5/9)}\text{”} \mapsto \text{“}\texttt{FtoC(temp)}\text{”}\}$, $\mathsf{PI} = \{\text{“}\mathsf{x} = \mathsf{x} + 1;\text{”}\}$, and $\mathsf{NI} = \emptyset$. 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 Feedback-driven semi-supervised synthesis

Require: Feedback oracle Feedback : $\mathcal{P} \times (\mathbb{T} \not\to \mathbb{T}) \times 2^{\mathbb{T}} \to (\mathbb{T} \to [-\infty, \infty])$.
Require: Semi-supervised synthesis engine SynthesisEngine.
Require: Input pool InputPool.
Require: Initial examples Examples, positive inputs PI, and negative inputs NI.

```
1:  while true do
2:      P ← SynthesisEngine(Examples, PI, NI)
3:      Rew ← Feedback(P, Examples, InputPool)
4:      (n, p) ← Thresholds(Examples, P, PI, NI, InputPool)
5:      PI′ ← {i ∈ InputPool | Rew(i) > p}
6:      NI′ ← {i ∈ InputPool | Rew(i) < n}
7:      if * then
8:          PI ← PI ∪ pi′ where pi′ is an arbitrary input from PI′
9:      else
10:         NI ← NI ∪ ni′ where ni′ is an arbitrary input from NI′
```

Algorithm 2 presents a procedure for the feedback-driven semi-supervised synthesis problem. 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, and an input pool InputPool, and produces a reward function Rew : InputPool $\rightarrow [-\infty, \infty]$. 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. This procedure is the one 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. The algorithm itself closely follows Figure 4. The only new component is the function Thresholds to choose p and n. The function Thresholds picks the thresholds based on all the information present in the current context. This function is dependant on the application scenario and the Feedback oracle. In Section 5, we demonstrate how this function was chosen in three different scenarios. The other major component of the procedure is the Feedback oracle. We present two different feedback oracles $\mathsf{Feedback}_{\mathsf{user}}$ and $\mathsf{Feedback}_{\mathsf{auto}}$. In the application scenarios, these oracles are combined in differently 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 of which can be converted to a reward function $\mathsf{Rew}_{\mathsf{U}}$: InputPool $\rightarrow [-\infty, +\infty]$. We have the following common cases (see Section 5):

- The user explicitly provides new positive inputs PI′ and negative inputs NI′. We convert this feedback into a reward function $\mathsf{Rew}_{\mathsf{U}}$ by setting $\forall$pi $\in$ PI.$\mathsf{Rew}_{\mathsf{U}}$(pi) = $+\infty$, $\forall$ni $\in$ NI.$\mathsf{Rew}_{\mathsf{U}}$(ni) = $-\infty$, and $\mathsf{Rew}_{\mathsf{U}}$(i) = 0 for all other inputs in InputPool.
- The user provides a set of candidate positive inputs PI*. For example, a set of candidate positive inputs could be a set of ASTs which are contain the cursor location in a file. Given PI*, we have $\forall$pi $\in$ PI*.$\mathsf{Rew}_{\mathsf{U}}$(pi) = x for some $0 < x < +\infty$.

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, the version control history, etc. Algorithm 3 presents an basic oracle that only reuses components from the semi-supervised synthesis engine, i.e., the provenance computation and the anti-unification. In practice, the feedback loop in Algorithm 2 can be optimized by sharing the provenance computation and anti-unification across the synthesis engine and $\mathsf{Feedback}_{\mathsf{auto}}$ oracle.

Algorithm 3 works as follows:

(a) If the program P on i produces an output that cannot be compiled, reward is $-\infty$,
(b) Otherwise, we synthesize a guard using $\mathsf{ReFazer}_{\mathsf{guard}}$ and collect all the inputs $\mathsf{InputPool}_{\mathsf{guard}}$ in the InputPool that satisfy the guard. If $\mathsf{InputPool}_{\mathsf{guard}}$ contains a significant fraction of InputPool, provide reward $-\infty$. Overly general guards are almost never the intended one.
(c) Otherwise, compute the distance between the i and an example input, using the Distance function. If this distance is high (resp. low), we give i a low (resp. high) reward.

We explain the Distance function informally: it is based on a combination of anti-unification and a clone detection technique (we use the technique from (Jiang et al., 2007)) that returns a distance between two trees.

Example 4.8. Consider the example `if(score < threshold)` $\mapsto$ `if(IsValid(score))`, and the candidate additional input `if(GetScore(run) < threshold)`. Say, a clone-detection technique on the two inputs returns some value d. Now, depending on threshold we have either the positive or negative case. Further, replacing `GetScore(run)` with larger and larger expressions will give greater and greater distances. However, from the program (and provenance information), we know that the

Algorithm 3 The Fully Automated Feedback Oracle $\text{Feedback}_{\text{user}}$

Require: Compiler Compiler : $t \to \mathbb{B}$ or $\perp$ if not compiler is available
Require: Distance metric Distance : $\mathbb{T} \times \mathbb{T} \to \mathbb{R}^{\geq 0}$
Require: Program $P = (P_{guard}, P_{trans})$
Require: Examples Examples : $\mathbb{T} \times \mathbb{T}$
Ensure: Rewards function Rew_E : InputPool $\not\to [-\infty, +\infty]$
1: $i^* \mapsto o^* \leftarrow$ arbitrary example in Examples
2: $\pi \leftarrow \text{Provenance}(i^* \mapsto o^*, P_{trans})$
3: $\text{Rew}_E \leftarrow \emptyset$
4: **for all** i ∈ InputPool **do**
5: **if** Compiler ≠ ⊥ ∧ Compiler(i) = false **then**
6: $\text{Rew}_E \leftarrow \text{Rew}_E \cup \{i \mapsto -\infty\}$
7: guard ← $\text{ReFazer}_{guard}(\{i \mid i \mapsto o \in \text{Examples}\} \cup \text{PI} \cup \{i\}, \text{NI}$
8: $\text{InputPool}_{guard} \leftarrow \{i \in \text{InputPool} \mid \text{guard}(i) = \text{true}\}$
9: **if** $\frac{|\text{InputPool}_{guard}|}{|\text{InputPool}|} > \epsilon$ **then** $\text{Rew}_E \leftarrow \text{Rew}_E \cup \{i \mapsto -\infty\}$ **continue**
10: $d \leftarrow \text{Distance}(i, i^*, \pi)$
11: $\text{Rew}_E \leftarrow \text{Rew}_E \cup \{i \mapsto (d < \text{threshold}) \; ? \; y : -y\}$
12: **return** Rew_E

internals of what expression appears in lieu of `score` does not matter to the transformation. Hence, we use anti-unification to produce the abstracted trees `if(h < threshold)` and `if(h < threshold)`, from i and i′ respectively, and then use the clone detection technique to measure distance. Here, it will return 0.

However, the technique from the above example may produce distances that are too low, and produce too many false positives. To avoid this problem, we use D-caps (Evans et al., 2009, Nguyen et al., 2013), a technique for identifying repetitive edits. Given a number $d >= 1$, a d-cap is a tree-like structure obtained by replacing all sub-trees of depth d with holes.

Example 4.9. Consider the following example (i ↦ o) ≡ `new Attribute(expression, Type, true)` ↦ `CreateSoftAttribute(expression, Type, true)` and a candidate positive input i_p ≡ `new Attribute(attributes)` that should not be accepted (they are calling different constructor of `Attribute`). Using anti-unification and clone-detection, we will assume that the distance is 0, as both inputs abstract to `newAttributes(h)`. On the other hand, if the `h` is replaced with it's 1-cap, the new abstractions are given `new Attribute(h1, h2, h3)` and `new Attribute(h)`. Now, the distance is not zero, with the right threshold, we would avoid this false positive.

5 APPLICATIONS OF SPARSE

In this section, we present three practical applications of Sparse 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 leverages 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 as inputs for the input pool.

5.1 Sparse $_{\text{loc}}$: User-Provided Additional Input Feedback

Sparse $_{\text{loc}}$ leverages user-driven feedback oracle to identify positive inputs to the semi-supervised synthesizer. The target for Sparse $_{\text{loc}}$ is applications where a user is providing examples manually.

To illustrate this application, consider our motivating scenario 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. Sparse $_{\text{loc}}$ uses the feedback to create a positive input and generalize the transformation. After that, Sparse $_{\text{loc}}$ 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, $\text{Feedback}_{\text{user}}$ returns a reward function Rew_U that is $+\infty$ on the additional positive input the user has provided, and is 0 everywhere else. Further, we pick the thresholds p and n to both have the value 0.

Sparse $_{\text{loc}}$ 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 Sparse that do not require explicitly providing examples and feedback.

5.2 Automated Feedback Based on Cursor Position (Sparse $_{\text{cur}}$)

For our second application, we instantiated the BluePencil algorithm (Miltner et al., 2019) using our semi-supervised synthesizer as the PBE synthesizer. Blue-Pencil 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 an 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 Sparse $_{\text{cur}}$ to learn transformations from just one example and one additional positive input.

To enable the completely modeless interaction, Sparse $_{\text{cur}}$ 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 user, 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 user 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 in the beginning of the line. There are many sub-trees 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 positive input. The $\text{Feedback}_{\text{user}}$ oracle returns reward function that gives a positive score (say x) to all sub-trees that include the position defined by the cursor. We use further feedback from $\text{Feedback}_{\text{auto}}$ oracle described in Section 4.2 to further disambiguate the cursor location. Intuitively, $\text{Feedback}_{\text{auto}}$ will provide positive rewards (say y) to all nodes that are "similar" to the example inputs. Finally, we pick thresholds n and p such that $n < x < p \wedge n < y < p$, but $x + y > p$.

We implement Sparse $_{\text{cur}}$ 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, Sparse $_{\text{cur}}$ 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 following section.

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

Our last application (Sparse $_{\text{auto}}$) uses a fully automated feedback to identify positive inputs without any explicitly or implicitly feedback from developers. The motivation for this application is that the

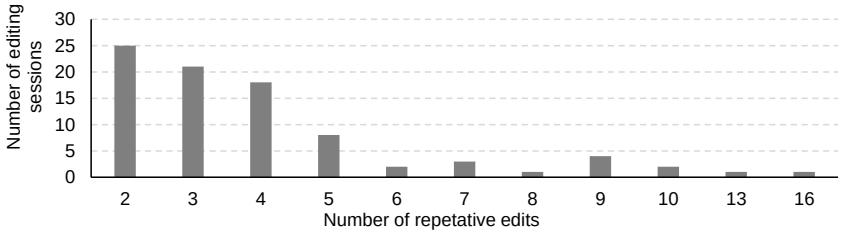

Fig. 6. The distribution of number of repetitive edits across the programs

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 SPARSE $_{\text{auto}}$ on top of Blue-Pencil. We restricted this application to synthesis tasks that have at least two input-output examples.

Consider our motivating example (Figure 1). As soon as the developer finishes the first two edits (Figure 1a), SPARSE $_{\text{auto}}$ automatically identifies the inputs in Figure 1c as positive inputs and synthesizes the correct transformation. Now, if the user is unaware of the other locations, the tool still produces suggestions at these places. These suggestions may then be used to automatically prompt the user to make these additional edits. Another scenario is as follows: after the two edits, the user 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 SPARSE with respect to the following research questions:

RQ1 *What is the effectiveness of the SPARSE* $_{\text{loc}}$ *in generating correct code transformations?*
RQ2 *What is the effectiveness of the reward calculation function?*
RQ3 *Given the cursor location, what is the effectiveness and efficiency of SPARSE* $_{\text{cur}}$*?*
RQ4 *What is the effectiveness and efficiency of feedback-driven synthesis SPARSE* $_{\text{auto}}$*?*

6.1 Benchmark

We collected 86 occurrences in real life code containing repetitive edits. These scenarios were collected from developers in a large company X spanning a team size of several thousand developers and millions of lines of code. We selected this sample from editing sessions with repetitive edits.

We construct a benchmark containing 86 program editing sessions. In each session, there is a list of program versions representing the program content before and after each edit. By comparing different program versions, we can easily figure out the *edit*. For each session, we manually generate a description containing the *number of edit*, the *version ids* before and after each edit and the *locations* where each edit is applied. Each editing session contains at least two repetitive edits and some noises (the edits that are not repetitive). 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%, program editing sessions with 2 repetitive edit. This percentage also motivates us to perform program synthesis with fewer examples, otherwise, we cannot generate any suggestions for such cases by learning from multiple examples. The averaged 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 one single program statement to large edits that modify code blocks.

Table 1. The effectiveness of semi-supervised synthesis.

Example	Session	Edit	Scenario	ReFazer		Sparse $_{loc}$	
				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%

6.2 Effectiveness of Sparse $_{loc}$

Suppose developers indicate the additional positive inputs that should be transformed, we first evaluate the effectiveness of Sparse $_{loc}$ by measuring its precision and recall in generating correct suggestions. In this evaluation, we use original ReFazer (Rolim et al., 2017), the existing program synthesis tool for code transformation, as our baseline approach.

6.2.1 Experimental setup. In each program edit session, we first extract all the repetitive edits from history versions. 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 can be used for testing, further, we limit N up to three. Considering users could perform the repetitive edits in any order, we consider all combinations when choosing the examples. For a 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. In each scenario, N edits are provided to PBE engine as examples, and one edit is used for testing. Therefore, for a editing session with M repetitive edits, suppose N examples are provided to the synthesis engine, we will create $C(M, N) * (M - N)$ scenarios. The Sparse $_{loc}$ also takes the testing edit as additional positive input. The synthesized transformation takes as input the testing edit inputs, and outputs the corresponding suggestions. We calculate the precision and recall of ReFazer and Sparse $_{loc}$ by measuring the number of false positives, false negatives, and true positives produced in all the scenarios.

6.2.2 Results. Table 1 presents our evaluation results of traditional ReFazer and Sparse $_{loc}$. Column *Example* presents the number of examples provided to PBE engine, while *Session* shows the number of program editing sessions. *Edit* and *Scenario* give the number of edits and scenarios, respectively. The more examples PBE engine takes, the more scenarios we create because there are more combinations when choosing examples. By comparing the different numbers 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 transformation easily results in false negatives. Especially, the recall with one example is just 26.71%, which also highlights the challenges to synthesize a high-quality transformation with fewer examples. In contrast, Sparse $_{loc}$ significantly improves the recall regardless of the number of examples, while keeps the high precision (slightly lower). Sparse $_{loc}$ can generate better results because the additional input could help synthesize a more properly generalized transformation. Specifically, we even achieve 100% recall and >96% precision with only one example, which can release the burden of users from providing multiple repetitive edit examples. Compare with ReFazer, we generate a few more false positives. The reasons will be discussed in discussion section.

Table 2. The effectiveness of the reward calculation function.

Session	# pNode	# nNode	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

SPARSE $_{loc}$ significantly improves the recall of REFAZER while keeps the high precision in generating correct suggestions. Even by taking one example as input, SPARSE $_{loc}$ 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. Reward calculation function determines whether a node is additional positive/negative input for the feedback system. In this section, we evaluate its effectiveness in identifying additional positive input by comparing with two baseline approaches: *No validation* and *clone detection*.

- *No validation*: regard any node as additional positive input;
- *Clone detection*: Given an edit $<i, o>$ and one node i_n, we determine whether i_n is an positive additional input by calculating the normalized distance between i_n and i using clone detection techniques. Here, we use the approach proposed by Jiang et al. (2007). If the distance is less than a threshold t (here, we set t as 0.3), i_n will be regarded as an additional positive input;
- *Reward function*: Given edit $<i, o>$ and node i_n, we use our proposed approach to calculate the reward score between $<i, o>$ and i_n. If the score is greater than threshold t (we set t as 0.7), i_n will be regarded as an valid additional positive input.

6.3.1 Experimental setup. Given an input i, we determine whether i is additional positive input using the above three different approaches. In each program editing session, we select the first edit as the example for the PBE engine, and remaining edit inputs as testing nodes. We mark the input of testing edit as *pNode*, which should be determined as additional positive input. To evaluate the precision, we also test our reward validation on irrelevant nodes, which is marked as *nNode*s. In each edit session, we regards all the nodes from the document that should not be transformed by the synthesized transformation as nNode. Once an input is determined as valid additional positive input, semi-supervised synthesis is invoked to synthesize suggestions for the additional inputs. We measure the false positive/negatives on both pNode and nNode produced by different approaches.

6.3.2 Results. Table 2 shows the evaluation results. By regarding any node as additional positive input, the synthesis engine can successfully generates suggestions for many of them. However, it also generates a large number of false positives (9055), which demonstrates the importance of the additional input validation. If we validate the additional input using existing clone detection (Column 6, 7), the false positive 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 *pNode*s, it 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 while do not introduce too many false negatives. Our reward calculation function results in 3 more false positive than clone detection. The underlying reason will be analyzed in the discussion section.

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

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

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

6.4 The Effectiveness and Efficiency of Sparse $_{cur}$

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

6.4.1 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 Sparse $_{cur}$. The history versions include at least one repetitive edit (e.g. e_1) and some irrelevant edits (noises). 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).

6.4.2 Result. Table 3 shows our evaluation result. Column *Scenario* 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, Sparse $_{cur}$ should be fast enough to ensure that the suggestion can be generated at run-time. Therefore, we also evaluate the efficiency of Sparse $_{cur}$ by measuring the time to generate each suggestion. Column *Time* describes the averaged time to generate edit suggestions. Basically, our engine could produce one suggestion in 51.8ms on average, and up to 441ms. At the cursor location, we believe generating suggestions in less than 0.44 second is acceptable.

Given one set of history versions and one cursor location, Sparse $_{cur}$ 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 The Effectiveness and Efficiency of Feedback-driven Synthesis Sparse $_{auto}$

In this experiment, we evaluate the efficiency and effectiveness of Sparse $_{auto}$, using automated feedback. Sparse $_{auto}$ should require fewer examples to synthesize a correct transformation. We measure the number of required input-output examples and additional inputs from developers to complete one edit task. In this evaluation, we use BluePencil as our baseline approach.

6.5.1 Experimental setup. Given a set of edits as edit task, developer needs to manually perform some edits, once synthesise engine can learn a correct transformation from the user-provided edits, it automates the remaining ones. For BluePencil, we assume developer manually produce versions $\{v_1, v_2, \ldots v_i\}$, and BluePencil automates the edits in the remaining versions $\{v_i \ldots v_n\}$. We measure the minimal number of repetitive edits in $\{v_1, v_2, \ldots v_i\}$ required from developers and the number of automatically generated edits. As for Sparse $_{auto}$, we also assume developer manually produce versions $\{v_1, v_2, \ldots v_j\}$. Meanwhile, we also assume developers provide their cursor locations.

Table 4. To complete all the repetitive edits, the required information (examples and locations) from users and the number of auto-generated suggestions. Column *Auto-input* is the number of additional inputs that are automatically identified by the feedback system. Column *Ratio* shows the averaged number of required examples to generate one suggestion.

Approach	Edit	Example	Location	Auto-input	Suggestion	Ratio	Time(s)
BluePencil	350	200	-	-	150	1.33	0.25
Sparse $_{auto}$	350	87	87	37	263	0.33	0.32

According the given history versions and cursor location, Sparse $_{auto}$ automatically founds more additional inputs, synthesizes program and automates the edits in the remaining versions $\{v_j \ldots v_n\}$. We measure the minimal number of repetitive edits in $\{v_1, v_2, \ldots v_j\}$ and the minimal number of cursor locations required from developers.

6.5.2 Results. Table 4 shows our evaluation results. Column *Edit* shows the total number of edits that need to be completed in the task. To complete all the edits, BluePencil requires users to manually perform 200 edits and it automates 150 edits (suggestions). On average, to generate one suggestion, users need to provide 1.33 (column *Ratio*) examples. Compared with BluePencil, Sparse $_{auto}$ just needs 87 examples and 87 user-indicated locations. Meanwhile, feedback system automatically find 37 additional inputs, shown in column *Auto-input*, that can improve the previously synthesized transformation. Corresponding, Sparse $_{auto}$ automatically produces more suggestions than BluePencil (263 vs 150). On average, it just needs 0.33 examples to produce one suggestion.

Meanwhile, Sparse $_{auto}$ should be fast enough to ensure that the suggestion 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* describes the averaged time to generate edit suggestions. Basically, our engine could complete the given edit task in 0.32 seconds on average. Compared to BluePencil, Sparse $_{auto}$ is a little bit slower because it continuously refines the transformation by invoking synthesis engine multiple times.

> To complete the same edit task, Sparse $_{auto}$ requires much fewer examples and generates more correct suggestions than BluePencil. On average, Sparse $_{auto}$ takes 0.32 seconds to complete one edit task.

6.6 Discussion

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

The semi-supervised synthesis produces a false positive in this case. Given the following edit: `Model(. . . , Outputs = null, Inputs = null)` ↦ `Model(. . . , Outputs = null)`, i.e., removing **Inputs=null**, and the following additional positive input `Model(. . . , Inputs = new List<ModelInput>), Outputs = null)` (note the switch of the last two parameters). Semi-supervised synthesis generates transformation that delete the last argument of `newMode` object creation. Therefore, the synthesized transformation will produce the suggestion for the additional input by deleting the last argument **Outputs = null**. However, the desired edit is delete 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 the anti-unification algorithm to handle commutativity (as the order of `name=value` style arguments is irrelevant. However, this changes

our anti-unification problem from AU to ACU (associativity-unity-commutativity), making the problem significantly more complex.

Limitations of the feedback oracles. In our experiment, SPARSE $_{\text{cur}}$ and SPARSE $_{\text{auto}}$ 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 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 too 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 selection of benchmark may not be representative for all types of edits developers. To reduce this threat, we collected real-world scenarios from the 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 SPARSE. To alleviate this issue, we also collected the traces of irrelevant edits and use them in our benchmarks. With respect to threats to construct validity, some scenarios of repetitive edits are ambiguous even for humans to understand what was the transformation intended by the developer, which may affect the construction of our benchmark. To reduce this threat, multiple authors of the paper reviewed the scenarios. If possible, we contacted the developer who made the edit for confirmation.

7 RELATED WORK

Interactive program synthesis. Interactive program synthesis systems allow users to incrementally refine semantic specifications 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. Instead of multiple programs, SPARSE can synthesize new programs using semi-supervised synthesis. Additionally, the technique not only leverages user feedback but also allows fully automated feedback during specification refinement. Our approach has the advantage that it allows users to refine intent with little to no modification to their workflow.

Semi-supervised learning. Semi-supervised learning 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 the 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, netting improvement to 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 the 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.

Toward this end, 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. The Getafix (Bader et al., 2019) and Revisar systems (Rolim et al., 2018) apply code repository mining techniques to discover such changes offline from large codebases, thus expanding breadth while also mitigating the burden for users to specify examples explicitly. Blue-Pencil (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 Blue-Pencil: the modeless interaction of Blue-Pencil 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. (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

Developer tools that proactively predict user's 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 rest of 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, Sparse 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.

REFERENCES

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.

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.

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.

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.

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.

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.

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.

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,

2015a.

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, 2015b.

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.

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.

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.

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.

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.