On the Fly Synthesis of Edit Suggestions

ANDERS MILTNER*, Princeton University, USA
SUMIT GULWANI, Microsoft, USA
VU LE, Microsoft, USA
ALAN LEUNG, Microsoft, USA
ARJUN RADHAKRISHNA, Microsoft, USA
GUSTAVO SOARES, Microsoft, USA
ASHISH TIWARI, Microsoft, USA
ABHISHEK UDUPA, Microsoft, USA

When working with a document, users often perform context-specific *repetitive edits* – changes to the document that are similar but specific to the contexts at their locations. Programming by demonstration/examples (PBD/PBE) systems automate these tasks by learning programs to perform the repetitive edits from demonstration or examples. However, PBD/PBE systems are not widely adopted, mainly because they require modal UIs – users must enter a special mode to give the demonstration/examples. This paper presents Blue-Pencil, a modeless system for synthesizing edit suggestions *on the fly*. Blue-Pencil observes users as they make changes to the document, silently identifies repetitive changes, and automatically suggests transformations that can apply at other locations. Blue-Pencil is parameterized – it allows the "plug-and-play" of different PBE engines to support different document types and different kinds of transformations. We demonstrate this parameterization by instantiating Blue-Pencil to several domains – C# and SQL code, markdown documents, and spreadsheets – using various existing PBE engines. Our evaluation on 37 code editing sessions shows that Blue-Pencil synthesized edit suggestions with a precision of 0.89 and a recall of 1.0, and took 199 ms to return suggestions on average. Finally, we report on several improvements based on feedback gleaned from a field study with professional programmers to investigate the use of Blue-Pencil during long code editing sessions. Blue-Pencil has been integrated with Visual Studio IntelliCode to power the *IntelliCode refactorings* feature.

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

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

ACM Reference Format:
Anders Miltner, Sumit Gulwani, Vu Le, Alan Leung, Arjun Radhakrishna, Gustavo Soares, Ashish Tiwari, and Abhishek Udupa. 2019. On the Fly Synthesis of Edit Suggestions. *Proc. ACM Program. Lang.* 3, OOPSLA, Article 143 (October 2019), 29 pages. https://doi.org/10.1145/3360569

*Anders Miltner performed this work as part of his internship with the Prose team at Microsoft.

Authors' addresses: Anders Miltner, Princeton University, USA, amiltner@cs.princeton.edu; Sumit Gulwani, Microsoft, USA, sumitg@microsoft.com; Vu Le, Microsoft, USA, levu@microsoft.com; Alan Leung, Microsoft, USA, alan.leung@microsoft.com; Arjun Radhakrishna, Microsoft, USA, arradha@microsoft.com; Gustavo Soares, Microsoft, USA, gustavo.soares@microsoft.com; Ashish Tiwari, Microsoft, USA, ashish.tiwari@microsoft.com; Abhishek Udupa, Microsoft, USA, abhishek.udupa@microsoft.com.

2475-1421/2019/10-ART143
https://doi.org/10.1145/3360569

1 INTRODUCTION

From source code to text files to slide decks, editing *documents* has become a pervasive component of many jobs. When working with a document for an extended period of time, users often must eventually perform *repetitive edits* – edits to the document that reflect a similar underlying change, but are performed at multiple locations within the document. For example, software programmers often have to deal with the task of performing repetitive code edits to add new features, refactor, and fix bugs during software development [Kim and Notkin 2009; Nguyen et al. 2013]. Word processor and presentation software users also usually make repetitive content and formatting changes such as updating all hyperlinks to a new server [Stack Exchange 2019] or changing the bullet/chart styles in the entire document [Edge et al. 2015].

Besides being tedious, manually performing these edits is error-prone and time consuming. When editing source code, programmers sometimes perform a single repetitive edit task over multiple editing sessions and over multiple commits as they miss places where the edit should have been applied, or they incorrectly apply the edit to some locations [Park et al. 2012; Rolim et al. 2017]. To reduce the user burden involved in making such mechanical edits, document editing tools implement *transformations* for some fixed classes of repetitive edits that are frequently encountered and universally applicable, such as ReSharper's [JetBrains 2019b] refactoring tools, PowerPoint's [Microsoft 2019b] snap-to features, and Microsoft Word's [Microsoft 2019c] autocorrect features.

Although the aforementioned tools help users apply general-purpose repetitive edits, they are not suited for edits that are only useful for a specific user and hence not candidates to be included in these tools. We call such edits *context-specific*[1]. The challenges of learning and applying context-specific repetitive edits are: (a) identifying *all* locations at which the edit must be applied, and (b) adapting the edit to the context of each of such locations (for instance, by renaming variables). Since each location differs syntactically, the programmer cannot perform a simple "find and replace" operation. To automate the context-specific repetitive edits, users normally have to write special scripts such as Microsoft Office macros [Microsoft 2019a], vim macros [Vim 2019], and Emacs' repeat-complex-command [GNU Emacs 2019]. This process is also error-prone, time consuming, and more importantly, out of reach for most non-expert end users.

Programming by demonstration/examples (PBD/PBE) has been applied to automate context-specific edits [Cypher 1993; Lau 2001; Lieberman 2001]. Recent advances in PBD/PBE allow users to synthesize edit scripts from demonstration or input-output examples in various domains such as text documents [Lau et al. 2001], source code [Meng et al. 2011, 2013; Rolim et al. 2017], and slide decks [Edge et al. 2015]. However, PBD/PBE systems have several problems that prevent their mass adoption in practice [Lau 2008]. First, they have modal UIs (i.e., users enter a special mode to give demonstration/examples), which interrupt users' workflow and require users to have knowledge about the systems to invoke them. Murphy-Hill et al. show that modal UIs are one of the greatest barriers to adoption for automated refactoring systems [Murphy-Hill and Black 2007; Murphy-Hill et al. 2009]. Second, PBD/PBE systems are domain-specific. A system usually targets one document domain and supports a certain kind of transformation. Third, PBD systems are sensitive to noise during demonstration. When users make a mistake, they usually have to start over to record a new trace. Finally, PBD/PBE systems do not provide *partial* solutions when they fail. In some cases, it would still be useful for the system to automate some of the repetitive edit instances, leaving the user to manually edit the remaining instances that could not be automatically performed.

Blue-Pencil. This paper presents Blue-Pencil, a system for suggesting repetitive document edits *on the fly*. Blue-Pencil observes users as they make changes to the document, silently

[1]We loosely define the current context to be current editing session. The context may thus include the function being edited, the file being edited, or the entire project, depending on how wide-ranging the user edits are.

identifies possible repetitive edits, and automatically suggests transformations that can be applied at other locations in the document. Learning repetitive edits on the fly without explicit input/output examples is challenging. We need to maintain the entire user edit history and distinguish the repetitive edits from large amounts of superfluous noise, such as non-repetitive edits and errors. Additionally, a sequence of smaller edits can be stacked together into a larger edit, which on one hand may be preferable because it reduces the number of suggestions to users, but on the other hand may cause the system to miss some potentially useful fine-grained suggestions. Blue-Pencil must decide whether a change is one completed edit, only a part of a larger edit, or in fact multiple disparate edits. Moreover, the synthesis algorithm must be fast, or Blue-Pencil would fail to suggest the edits before the user simply completes the edit manually.

At a high level, Blue-Pencil maintains the edit history as a directed acyclic multigraph (DAM) whose nodes are document versions and edges are edits between two versions. As a user is making changes, Blue-Pencil updates the DAM accordingly and tries to identify similar edits by comparing the edges. To avoid comparing all possible combinations of edges, Blue-Pencil uses nearest-neighbour-based heuristics to identify likely candidates. Blue-Pencil then invokes two respective PBE engines on the candidates to synthesize (1) the guard, which determines the locations which should be transformed, and (2) the transforming program to apply on the parts of the document identified by the guard. As there are potentially many suggestions (due to different candidates, different guards, and different transforming programs), Blue-Pencil uses a ranking system to pick suggestions that best explain the edits performed by the user.

Blue-Pencil's parameterization enables it to "plug-and-play" different PBE engines to support different document types (e.g., source code, markdown documents, and spreadsheets) and different kinds of transformations (e.g., tree-based and string-based transformations). Our system is *modeless*: users do not explicitly enter a special mode to give demonstration or examples. Instead, the intent is inferred automatically, thus avoiding discoverability and context-switching problems. Blue-Pencil also allows users to make repetitive changes in arbitrary manners, which makes it more flexible than traditional PBE/PBD systems. For instance, users can perform the second instance of the a repetitive edit either by moving text from the first edit with a copy-paste command and changing the relevant parts, or by typing the full change manually. Furthermore, because Blue-Pencil maintains the edits at various granularities, even if a bigger change is not expressible in the underlying language of transformation programs, Blue-Pencil may still be able to suggest smaller repetitive edits that partially automate the task. In contrast, traditional PBE/PBD systems require users to break down the task to the right level of abstraction; if they do not, the systems fail completely.

Instantiations and evaluation. To demonstrate the benefits of Blue-Pencil's parameterization, we instantiated Blue-Pencil to several domains: C# and SQL code transformation using Refazer [Rolim et al. 2017] as the underlying PBE system, and markdown document and spreadsheet transformation using a combination of Refazer [Rolim et al. 2017], FlashFill [Gulwani 2011], and FlashProfile [Padhi et al. 2018].

We evaluated the performance and accuracy of Blue-Pencil using the C# and SQL instantiation. We simulated the use of Blue-Pencil on 37 fine-grained edit histories obtained by logging two external programmers and one of the authors. In our experiments, Blue-Pencil identified correctly all repetitive edits (recall 1.0) with average suggestion time of 199 ms. Some of Blue-Pencil suggestions are false positives, but they happened when the user was in the middle of typing the repetitive edit and disappeared as soon as the user finished the edit. The overall precision of Blue-Pencil was 0.89.

We also collected additional qualitative feedback via an in-situ field study to investigate the effectiveness of Blue-Pencil with a team of over 25 programmers. The sentiment in the collected

feedback was generally positive, with a few minor issues raised. We used this feedback to improve the usability, accuracy, and performance of Blue-Pencil, which forms the basis for the *IntelliCode refactorings* feature in Visual Studio IntelliCode[2].

Contributions. This paper makes the following contributions:

- We formalize the problem of synthesizing repetitive edit suggestions on the fly (§3).
- We propose Blue-Pencil and its algorithm for automatically suggesting repetitive edits on the fly (§5). To the best of our knowledge, this is the first system of its kind.
- We demonstrate that by supplying Blue-Pencil with different PBE engines we can apply the technique to several document domains with different kinds of transformations (§6).
- We evaluated Blue-Pencil on a suite of 37 real code edit histories. The results suggest that Blue-Pencil can correctly automate real repetitive edit tasks without requiring explicit examples within acceptable time (§7).
- We evaluated Blue-Pencil in a real programming environment with an in-situ field study (§7.6).

2 REPETITIVE EDITS AND WHY THEY SHOULD BE AUTOMATED.

We now illustrate the challenges and subtleties of the on the fly edit suggestion problem. Our motivating example is inspired by a real scenario where Blue-Pencil helped a programmer perform repetitive edits during our field study (Section 7.6). Section 6 provides more examples from other domains.

2.1 Setting and Problem

Figure 1 shows a simplified history of code edits a programmer made to filter forbidden attributes for several nodes. Each state in the history represents a parsable version of the code. Initially, the code is in version v_0. The programmer performs the following changes to the document:

(1) In line 730, she adds a `Where` clause to obtain only *valid* attributes specific to the node (version v_{14}). Prior to this edit are 13 other parsable edits (omitted from the figure). Notice that although version v_{14} is parsable, the code does not compile because the method `IsValid` does not exist.
(2) In lines 741-744, she adds the method `IsValid` (version v_{48}). At this point, the code compiles.
(3) The programmer notices that she needs to update the code in line 735 as well, perhaps because it is close to the first change in line 730 (version v_{55}).
(4) The programmer thinks that she is done but Blue-Pencil, which is running in the background and detecting the repeated patterns, alerts that there are two other locations where the changes should have been applied (the blue squiggles in lines 312 and 579 – version v_{55}). Missing these locations would result in a bug in the code.
(5) The programmer agrees and accepts the suggestions (version v_{57}).

Notice that the above repetitive edits were context-specific, and thus would be unsuitable for inclusion in a refactoring tool.

Why are Repetitive Edits irksome? Performing repetitive edits manually is clearly tedious. Apart from tedium, a programmer might forget to perform the repetitive edit at some locations. While some locations can be detected because missing them leads to compilation error, this "compile-edit" cycle is not only tedious but also potentially time consuming in large projects with significant compilation times. In the worst case, a missed edit remains undetected by the compiler, leading to a bug in the code – indeed, this would be the case in our motivating example.

[2]https://devblogs.microsoft.com/visualstudio/refactoring-made-easy-with-intellicode/

v_0

```
...
312    var attrs = child.Attrs();
...
579    var attrs = selectedNode.Attrs();
...
730    var attrs = node.Attrs();
...
735    var parentAttrs = node.Parent.Attrs();
...
```

v_{14}

```
...
312    var attrs = child.Attrs();
...
579    var attrs = selectedNode.Attrs();
...
730    var attrs = node.Attrs().Where(l => IsValid(node.Kind, l.Name));
...
735    var parentAttrs = node.Parent.Attrs();
...
```

v_{48}

```
...
312    var attrs = child.Attrs();
...
579    var attrs = selectedNode.Attrs();
...
730    var attrs = node.Attrs().Where(l => IsValid(node.Kind, l.Name));
...
735    var parentAttrs = node.Parent.Attrs();
...
735 }
...
741 private bool IsValid(string kind, string attr){
...    // check if attr is valid for kind
744 }
```

v_{55}

```
...
312    var attrs = child.Attrs();
...
579    var attrs = selectedNode.Attrs();
...
730    var attrs = node.Attrs().Where(l => IsValid(node.Kind, l.Name));
...
735    var parentAttrs = node.Parent.Attrs().Where(l => IsValid(node.Parent.Kind, l.Name));
...
```

v_{57}

```
...
312    var attrs = child.Attrs().Where(l => IsValid(child.Kind, l.Name));
...
579    var attrs = selectedNode.Attrs().Where(l => IsValid(selectedNode.Kind, l.Name));
...
730    var attrs = node.Attrs().Where(l => IsValid(node.Kind, l.Name));
...
735    var parentAttrs = node.Parent.Attrs().Where(l => IsValid(node.Parent.Kind, l.Name));
...
```

Fig. 1. A simplified history of code changes that contains both programmer edits and Blue-Pencil's suggestions.

Can we use Programming-by-Example (PBE) to solve this? Earlier work [Meng et al. 2013; Rolim et al. 2017] has proposed the use of PBE techniques to automate code refactoring. These techniques can indeed be used to automate repetitive edits, and our proposed approach uses earlier work as a component. However, the primary limitation of these prior approaches [Meng et al. 2013; Rolim et al. 2017] is that they require explicit specification of input-output examples.

In the context of our example in Figure 1, using PBE requires that the programmer (a) be aware of the existence of the feature and how to invoke it, (b) think ahead about the repetitive changes he plans to make and explicitly provide examples as pairs of versions (v_0, v_{14}) and (v_{48}, v_{55}), and (c) remember to switch to a special mode during which to provide those examples. This series of interactions is problematic, as step (a) suffers from the *discoverability problem* [Murphy-Hill and Black 2007; Murphy-Hill et al. 2009], and steps (b) and (c) both intrude on the programmer's normal workflow.

Blue-Pencil: Non-intrusively watch, learn, and make intelligent suggestions. Blue-Pencil aims to be both *modeless* and *proactive*. Blue-Pencil is *modeless* in that a user editing a document need not switch modes to *explicitly* provide input-output examples that describe the intended repetitive edit. Instead, Blue-Pencil *automatically infers* input-output examples by observing the user's changes to the document over time. Once an appropriate set of input-output examples has been identified, Blue-Pencil then leverages an appropriate PBE engine to synthesize a program that realizes the intent captured in the inferred examples. Blue-Pencil is *proactive* in that it identifies other locations in the document where the (inferred) repetitive edit is applicable and offers to automatically apply the edit on the user's behalf. Returning to our example, the net effect is that by observing the sequence of document versions $v_0, \ldots, v_{57}$, Blue-Pencil offers the programmer the option of automatically applying the repetitive edit at lines 312 and 579.

Concretely, Blue-Pencil aims to identify (and abstract) repetitive edits from a series of *document versions* — which we refer to as an *edit history*. There are several challenges that must be overcome for Blue-Pencil to produce correct (and useful) suggestions, which we now discuss.

2.2 Blue-Pencil: The Challenges

Noise and unrelated edits. The changes to line 730 (v_{14}) and line 735 (v_{55}) are two separate instances of the same repetitive edit (addition of a `Where` clause of specific form). However, note that the user has interleaved those changes with a non-repetitive change to lines 741-744 (v_{48}). There are two important characteristics to note. First, a single repetitive edit instance can span across several versions in the edit history (the first instance spans 15 individual versions, from v_0 to v_{14}). Second, different instances of a repetitive edit need not occur consecutively in the edit history: the changes to lines 730 and 735 are separated by a non-repetitive change adding the `IsValid` method definition. Despite the presence of such noise, Blue-Pencil should still be able to identify the repetitive edits.

Transient edits. In the process of inserting the `Where` method call at line 730 in version v_{14}, the programmer types a sequence of prefixes of the string "W", "Wh", ..., "Where". Although not shown in Figure 1, versions of the program with each of these prefixes will likely be part of the edit history provided to Blue-Pencil. To avoid producing spurious suggestions that include all of these prefixes, Blue-Pencil must identify and ignore these *transient* edits.

Identifying similar edits applied in different ways. The repetitive edit shown in Figure 1 consists of many fine-grained edits. For example, the edit in line 730 consists of 14 individual fine-grained edits from v_0 to v_{14}. The analogous change to line 735 consists of 7 fine-grained edits from v_{48} to v_{55}. Although they represent two instances of the same repetitive edit, they were applied in different

ways: while the programmer typed the entire `Where` expression in line 730, she instead modified line 735 by copying and pasting the expression from line 730 and inserting the string "`.Parent`". Blue-Pencil must be able to identify similar edits even when they are applied in different ways.

Generalizing user intent. The artifacts used to infer user intent in Programming-by-example or programming-by-demonstration systems are inherently ambiguous. For instance, given the first two instances of the repetitive edit in lines 730 and 735, one option would be to generalize them to the transformation that adds the `Where` clause to the end of every expression containing the object named `node`. Another option would be to generalize to obtain the transformation that adds the `Where` clause to every expression of the form `_.Attrs()` that is assigned to a fresh variable created using `var`. The options illustrate the complexity of finding the right generalization from examples that is common to all PBE systems.

Additionally, the modeless, on the fly scenario in Blue-Pencil brings further ambiguity. In a lot of cases, it is unclear which of the programmer's edits should be generalized into one single transformation, and at what granularity. For example, if in addition to the changes at line 730 and line 735, suppose the programmer also changed the `var` in each line to `IEnumerable<Attribute>`. Now, the granularity of edits add further ambiguity—is the right generalization the full transformation of adding the `Where` clause and specifying the type of the variable? Or is the right generalization to treat these changes as two separate repetitive transformations, one to add the `Where` clause and one to specify the type? The choice comes with trade-offs: treating the changes as one single repetitive edit makes the suggestion not applicable to cases where the result of `_.Attrs()` is not assigned to a fresh variable, while treating them as two separate repetitive edits make the suggestions less useful and potentially noisy.

2.3 Repetitive Edits: Beyond Code

We have used an example involving code transformations to illustrate the repetitive edit problem and to highlight the desiderata for any solution. However, we emphasize that repetitive edits naturally arise in other activities which involve creation of content. Blue-Pencil is *parameterized*, in that it is agnostic to the underlying document class as long as relevant PBE engines for by-example synthesis are available. Leveraging this parameterization and commonly available PBE engines, we apply Blue-Pencil to automate repetitive edits in markdown documents and in spreadsheets. See Section 6 for a full description of these instantiations.

3 OPTIMAL EXPLANATION GENERATION PROBLEM

We first formalize the *optimal explanation generation* problem, which we use as a way to perform *on-the-fly edit suggestion.*

3.1 Documents, Versions, and Edits

Documents. We use the term *document* to refer to any entity that a user edits and modifies over time. We use the notation $\mathbb{D}$ to denote a class of documents. We use the notation $v \in \mathbb{D}$ for an individual document, and we call each new document produced as the user edits a *version*. Examples of document classes include: (a) abstract syntax trees, (b) Markdown (or other structured text) documents, (c) spreadsheets, etc.

Locations and Lenses. We associate every change to a document with a "location". Informally, an edit to the document only modifies the contents at a single location: each edit (a) retrieves the contents of a single location, (b) changes the retrieved contents, and (c) updates the document by putting the changed contents into the given location. Examples of such locations include an individual sub-tree of an AST, a line in a flat text document, a single paragraph in a structured

text document, a single column in a spreadsheet, etc. Locations are not necessarily disjoint—for example, two sub-trees of an AST have a non-empty intersection if one is an ancestor of the other.

We use *lenses* to formalize of the concept of locations [Foster et al. 2007]. Formally, a document class $\mathbb{D}$ is associated with a universe of lenses $\mathbb{L}$. Each lens $\ell \in \mathbb{L}$ consists of a *getter* $\ell_{get} : \mathbb{D} \mapsto \mathbb{V}$ which retrieves some view (of some part) of the document, and a *setter* $\ell_{set} : \mathbb{D} \times \mathbb{V} \mapsto \mathbb{D}$ which updates the same part of the document that is retrieved by the getter to produce a new version of the document.

Example 3.1. In our canonical application, the universe of all documents $\mathbb{D}$ is the set of all abstract syntax trees (ASTs). Consider the edit shown in line 730 (Figure 1), where the programmer adds the `Where` expression to the existing `Attrs` expression. This specific operation on a given document (AST) v is modeled as: (a) selecting the sub-tree s of the AST that corresponds to the call `node.Attrs()`, (b) creating a new sub-tree s' by cloning s and adding the nodes for the `Where` clause, (c) updating the given AST v by replacing s with s'. The lens ℓ corresponds to the location of the sub-tree s for the method call `node.Attrs()`; ℓ_{get} retrieves s and ℓ_{set} replaces s with any updated contents. Here, the view type $\mathbb{V}$ of the lens is the set of sub-trees of the AST.

Remark 3.1. *In our canonical setting of source code edits, we make the choice that our techniques operate on ASTs rather than source code text. This choice disallows Blue-Pencil from learning certain kinds of edits: for example, we cannot fix syntax errors.*

Edits and Histories. An *edit* is a pair of documents (v_b, v_a) where v_b and v_a are the documents *before* and *after* the edit, respectively. A *history* is a sequence of documents $h = v_0 v_1 \ldots v_n$ created by a sequence of edits. For the history h, we define: (a) *Fine-grained edits* in h as $\mathsf{FineEdits}(h) = \{(v_i, v_{i+1}) \mid 0 \leq i < n\}$, and (b) *Transitive edits* in h as $\mathsf{TransEdits}(h) = \{(v_i, v_j) \mid 0 \leq i < j \leq n\}$.

3.2 Explaining Edit Histories

Programs for Edit Explanation. We define *programs* for explaining edits to a document. Informally, we want multiple edits that are different instances of the same repetitive change to be explained by the same program.

A *program* $\mathsf{P} : \mathbb{D} \to 2^{\mathbb{D}}$ transforms a document into (zero or more) new versions of that document. A program P is defined by a pair $(\mathtt{guard}_\mathsf{P}, \mathtt{trans}_\mathsf{P})$, where $\mathtt{guard}_\mathsf{P} : \mathbb{D} \to 2^{\mathbb{L}}$ and $\mathtt{trans}_\mathsf{P} : \mathbb{V} \to \mathbb{V}$. The semantics of P is defined as follows: $\mathsf{P}(v) = \{v' \mid \exists \ell \in \mathtt{guard}_\mathsf{P}(v) : v' = \ell_{set}(v, \mathtt{trans}_\mathsf{P}(\ell_{get}(v)))\}$. Intuitively, P changes the part of the document provided by the lenses returned by $\mathtt{guard}_\mathsf{P}$ using the transformation $\mathtt{trans}_\mathsf{P}$, i.e., each document in the set returned by P has the change applied to a single relevant location. The set of all programs is denoted by $\mathbb{P}$. A program P *explains* an edit (v, v'), denoted as $v \rightarrow_\mathsf{P} v'$, if $v' \in \mathsf{P}(v)$.

Example 3.2. Consider the scenario in Figure 1. All of the edits in lines 312, 579, 730, and 735 can be explained by the same program $(\mathtt{guard}_\mathsf{P}, \mathtt{trans}_\mathsf{P})$ where:

- $\mathtt{guard}_\mathsf{P}(v)$ returns the lenses corresponding to sub-trees of form *target*`.Attrs()` in v, where *target* is some method call target. In the AST encoding used in our experiments, they are sub-trees having the label InvocationExpression and contain two children: one for the method call target and one for the function application `Attrs()`.
- $\mathtt{trans}_\mathsf{P}(s)$ takes as input the sub-tree s (returned by $\mathtt{guard}_\mathsf{P}$) and transform it into s' that has the `Where` clause addition. In our encoding, s' is also a InvocationExpression node having two children: the first child represents *target*`.Attrs()` and the second one is the `Where` clause. In addition to creating new nodes, Blue-Pencil also copies nodes from s (e.g., *target*`.Attrs()` and the variables inside the lambda) to generate context-specific edits.

Program Sets for History Explanation. A *program set* $\mathsf{SSuite} \subseteq \mathbb{P}$ is a set of programs. A program set $\mathsf{SSuite} \subset \mathbb{P}$ is said to *explain* a history $v_0 v_1 \dots v_k$ if the following two conditions hold:

(a) there is a sub-sequence $v'_0, v'_1, \dots, v'_m$ of the history with $v'_0 = v_0$ and $v'_m = v_k$, and a sequence of (not necessarily distinct) programs $\mathsf{P}_0, \mathsf{P}_1, \dots \mathsf{P}_{m-1}$ from SSuite such that each edit (v'_j, v'_{j+1}) is explained by P_j. Explicitly, we want: $v_0 = v'_0 \rightarrow_{\mathsf{P}_0} v'_1 \rightarrow_{\mathsf{P}_1} v'_2 \rightarrow_{\mathsf{P}_2} \cdots \rightarrow_{\mathsf{P}_{m-2}} v'_{m-1} \rightarrow_{\mathsf{P}_{m-1}} v'_m = v_k$; and

(b) each explanation $v'_j \rightarrow_{\mathsf{P}_j} v'_{j+1}$ is either:
- *Repetitive*: program P_j repeats, i.e., $\exists j' \neq j : \mathsf{P}_j = \mathsf{P}_{j'}$, or
- *Fine-grained non-repetitive*: program P_j does not repeat and (v'_j, v'_{j+1}) is a fine-grained edit in the history. Formally, $\nexists j' \neq j : \mathsf{P}_j = \mathsf{P}_{j'} \implies \exists k. v'_j = v_k \wedge v'_{j+1} = v_{k+1}$.

In a nutshell, a program set explains a history if there is a path from the initial document version to the final document version, where each edge is labelled by a program. These edges have the restriction that if a program only labels one edge, the edge must go from version i to version $i+1$. Formalizing the program set as part of a path ensures the explanations are well-formed – each edit must be taken into account exactly once. Restricting the non-repetitive edits to only apply to fine-grained edits lets optimal program sets simply be the smallest program sets.

Example 3.3. Consider the history $v_0 v_1 \dots v_{54} v_{55}$ in Figure 1. One explanation of the history is given by $v_0 \rightarrow_{\mathsf{P}} v_{14} \rightarrow_{\perp_{14}} v_{15} \rightarrow_{\perp_{15}} v_{16} \cdots \rightarrow_{\perp_{47}} v_{48} \rightarrow_{\mathsf{P}} v_{55}$, where:
- P is a *repetitive* program that selects method calls of form *target*.`Attrs()` and adds `Where` expressions (Example 3.2).
- $\perp_{14}, \perp_{15}, \cdots$, and $\perp_{47}$ are *non-repetitive* programs that only convert v_{14} to v_{15}, v_{15} to $v_{16}, \cdots$, and v_{47} to v_{48}, respectively.

P can skip over versions such as v_2 in the history because it is repetitive while other non-repetitive programs such as $\perp_{14}$ to $\perp_{47}$ are only allowed to explain fine-grained edits.

For any given history, there may exist a large number of explanations. Informally, we would like to synthesize the explanation that represents the user intent. Intuitively, simpler explanations tend to better represent user intent [Gulwani 2011]. For instance, when comparing a single program that explains a repetitive edit in a location to multiple consecutive programs that explain smaller repetitive edits to sub-locations of this location, it is more likely that the single program represents the user intent. Our approach for predicting user's future edits is predicated on the hypothesis that **a good explanation for the user's past edits can help predict the user's future actions**. In particular, if SSuite explains history $H = d_0 d_1 \dots d_k$, then the applications of the programs in SSuite on the final version d_k will give us possible next edits of interest.

We thus aim at generating an explanation with as many edits as possible being explained by as few repetitive programs as possible. An explanation SSuite for a edit history h is *optimal* if $|\mathsf{SSuite}| \leq |\mathsf{SSuite}'|$ for any other explanation SSuite′ for h. The below problem statement asks for this property.

Definition 3.4 (Optimal Explanation Generation). Given a history h and the set of all programs $\mathbb{P}$, the *optimal explanation generation* problem is to produce a program suite $\mathsf{SSuite} \subseteq \mathbb{P}$ that optimally explains h.

Remark 3.2 (Alternative ways to predict user edits.). *In our setting, the suggestion system is not clairvoyant to new code the programmer might write, so we restrict ourselves to suggesting repetitive edits. Note that this is unlike some existing works which may suggest edits that do not come from the history, by learning from common edits in large source code corpora [Foster et al. 2012; Negara et al. 2014; Raychev et al. 2014; Yin et al. 2019].*

Algorithm 1 Oracle-based Explanation Generation Algorithm

```
1: function OracleBasedExplanationGenerator(h = v_0 v_1 ··· v_k)
2:     Edits ← FineEdits(h) ∪ TransEdits(h)
3:     (V, E) ← ({v_0, ..., v_k}, ∅)
4:     for SubEdits ∈ 2^Edits do                          ▷ Foreach subset of Edits
5:         if |SubEdits| ≤ 1 ∨ Overlap(SubEdits) then
6:             continue
7:         label ← Oracle(SubEdits)                       ▷ Ask the Oracle for explanation
8:         if label ≠ ⊥ then          ▷ A common explanation found for edits in SubEdits
9:             E ← E ∪ {(v_a, label, v_b) | (v_a, v_b) ∈ SubEdits}   ▷ Add edges labeled with label
10:    E ← E ∪ {(v_i, ⊥_i, v_{i+1}) | 0 ≤ i < k}          ▷ Add non-repetitive explanations labeled ⊥_i
11:    return LeastColorfulValidPath(V, E, v_0, v_k)
```

4 ORACLE-GUIDED APPROACH FOR OPTIMAL EXPLANATION GENERATION

Given a history $h = v_0 v_1 \cdots v_k$ of edits on some document, in this section, we describe a procedure to find an optimal explanation for h. The procedure uses an oracle that generates explanations of subset of edits from the history over some space $\mathbb{P}$ of programs.

Let $\texttt{Edits} = \texttt{FineEdits}(h) \cup \texttt{TransEdits}(h)$ be the union of all fine-grained edits and transitive edits obtained from the history h. Note that $|\texttt{Edits}| = k(k+1)/2$. We assume that we have access to an oracle Oracle that given a subset $\texttt{SubEdits} \subseteq \texttt{Edits}$ of edits, returns a program p from some $\mathbb{P}$ that explains *all the edits* in SubEdits. We say Oracle is *correct* if (a) Oracle(SubEdits) returns $p \in \mathbb{P}$ if and only if p explains every edit in SubEdits and (b) it returns $\perp$ if and only if no such $p \in \mathbb{P}$ exists.

Two edits $(v_i, v_{i'})$ and $(v_j, v_{j'})$ in Edits are said to *overlap* if $[i, i'-1] \cap [j, j'-1] \neq \emptyset$. A subset SubEdits of (fine-grained and transitive) edits *overlaps* if it contains two overlapping edits. Let Overlap(SubEdits) return true if and only if SubEdits is overlapping.

Algorithm 1 uses Oracle to find an optimal explanation for a given history. There are two steps in the algorithm. The first step constructs a directed acyclic multigraph (DAM) whose nodes represent the document versions present in the given history, and edges represent edits. Edges are labeled by an explanation for that edit. There can be multiple edges between the nodes. Construction of the DAM involves enumerating all possible sets of (non-overlapping) edits and using the oracle to find a common explanation for that set of edits. Each fine-grained edit (v_i, v_{i+1}) is also labeled by a default explanation $\perp_i$. Informally, $\perp_i$ is a program such that $\perp_i(v_i) = \{v_{i+1}\}$, and $\perp_i(v) = \emptyset$ for all $v \neq v_i$. The second step computes an optimal explanation by finding the least colorful valid path from v_0 to v_k in the directed acyclic multigraph. Here, the color of an edge (v_a, label, v_b) in the DAM is the program label, and a valid path is one where each color that is not $\perp_i$ for some i appears more than once. The $\texttt{LeastColorfulValidPath}(V, E, v_0, v_k)$ procedure returns the valid path in the DAM that has the fewest distinct colors along its edges. Algorithm 1 can be shown to be sound and complete, modulo the correctness of the oracle.

Theorem 4.1. *Assuming that the oracle is correct with respect to some underlying program set $\mathbb{P}$, the collection of all labels in the path from v_0 to v_k returned by* $\texttt{OracleBasedExplanationGenerator}(h)$ *is an optimal explanation for the history h over the program set $\mathbb{P}$.*

Proof. First, we show that there is a correspondence between valid paths in the DAM and explanations. Let $\{\mathsf{P}'_0, \ldots, \mathsf{P}'_{n-1}\}$ be an explanation of h and let $v'_0 \to_{\mathsf{P}'_0} v'_1 \to_{\mathsf{P}'_1} \ldots \to_{\mathsf{P}'_{n-1}} v'_n$ be the witness. Recall that not all P_i are distinct.

- For a repetitive P'_i, let $\{(v_{i_1}, v_{i_1+1}), \ldots, (v_{i_m}, v_{i_m+1})\}$ be the set of edits explained by P_i. In line 4, the procedure iterates over every subset of edits in the history, and calls Oracle on the subset. When SubEdits $= \{(v_{i_1}, v_{i_1+1}), \ldots, (v_{i_m}, v_{i_m+1})\}$, by soundness of Oracle, we have that Oracle(SubEdits) $= \mathsf{P}_i \neq \bot$. Hence, the edges $(v_{i_j}, \mathsf{P}_i, v_{i_j+1})$ are added to E. Note that P_i may not be the same as P'_i if there is more than one program in $\mathbb{P}$ that explains each edit in SubEdits. However, if $\mathsf{P}'_i = \mathsf{P}'_j$, then we have that $\mathsf{P}_i = \mathsf{P}_j$.
- For a non-repetitive P'_i, we have that (v'_i, v'_{i+1}) is a fine-grained edit. Hence, the edge $(v'_i, \bot_i, v'_{i+1})$ is added to E in line 10. Let $\mathsf{P}_i = \bot_i$.

Now, the path $(v'_0, \mathsf{P}_0, v'_1), \ldots, (v'_{n-1}, \mathsf{P}_{n-1}, v'_n)$ is a valid path in the DAM from v_0 to v_n. Further, by construction, the number of colors along this path is at most the size of the explanation due to the implication $\mathsf{P}'_i = \mathsf{P}'_j \implies \mathsf{P}_i = \mathsf{P}_j$.

In the reverse direction, given a valid path $(v'_0, \mathsf{P}_0, v'_1), \ldots, (v'_{n-1}, \mathsf{P}_{n-1}, v'_n)$ in the DAM, we can construct the explanation $\{\mathsf{P}_0, \ldots, \mathsf{P}_{n-1}\}$ with the witness $v'_0 \rightarrow_{\mathsf{P}_0} v'_1 \ldots \rightarrow_{\mathsf{P}_{n-1}} v'_n$. The validity of the path enforces that each P_i either repeats multiple times or is equal to $\bot_i$ for some i. Hence, each non-repetitive program in the explanation is a fine-grained edit, implying the validity of the witness. Here, the size of the explanation is equal to the number of colors in the path.

Given the above correspondence between valid paths and explanations (along with the relations between the number of colors in a valid path and the size of the corresponding explanation), we get that the set of colors in the least colorful valid path is a valid optimal explanation. □

While Algorithm 1 is correct, the complexity of the procedure is high, even modulo the oracle. In the first step of the procedure, we make an exponential number of calls to the oracle in the length k of the history. Furthermore, the problem of finding a least colorful valid path in a directed acyclic multigraph, which is used in the second step, is easily shown to be NP-hard. We, therefore, use heuristics inspired from machine learning and greedy approximation schemes to obtain a practical explanation generation procedure.

5 ON-THE-FLY EXPLANATION GENERATION

Our motivation for generating an explanation for a history of edits is to use the explanation to provide suggestions to the user about possible future edits *as the user is making the edits*. In this section, we present an incremental and efficient procedure for explanation generation that follows the logical flow of the OracleBasedExplanationGenerator procedure.

The OracleBasedExplanationGenerator procedure contains three main ingredients: (a) an oracle that returns a program that explains a given set of (non-overlapping) edits, (b) an exhaustive enumeration based procedure for creating the DAM, and (c) a solver for the least colorful path problem on the DAM. The oracle is implemented using programming-by-example (PBE) technology. The process for creating the DAM is optimized by eliminating transient versions, and defining a distance metric on edits – based on featurization of edits – and using proximity to suggest the subsets of edits that are likely to have a (common) explanation. Finally, the approach for least colorful path is based on using a greedy heuristic. We describe these three in detail next.

5.1 Using PBE to implement the oracle

Program synthesis by example (PBE) is concerned with synthesizing programs given a partial specification in the form of a few input-output examples. The key challenge that PBE engines solve is that of generalization: the synthesizer has to generalize from the given input-output examples to learn the transformation intended by the user. However, the synthesizer should not over-generalize for it might then learn a transformation that the user views as being unsound.

Algorithm 2 PBE based implementation of the Oracle.

Require: DSL for guards: DSL_G
Require: DSL for transformations: DSL_T
1: **function** PBEOracle($\{(v_{b_1}, v_{a_1}), \ldots, (v_{b_n}, v_{a_n})\}$)
2: GuardSpec = $\{v_{b_i} \mapsto \text{DiffLoc}(v_{b_i}, v_{a_i}) \mid 1 \leq i \leq n\}$
3: ▷ Differing locations (lenses) from each edit are examples for guard
4: TransSpec = $\{\ell^i_{get}(v_{b_i}) \mapsto \ell^i_{get}(v_{a_i}) \mid 1 \leq i \leq n \wedge \ell^i = \text{GuardSpec}(v_{b_i})\}$
5: ▷ The change at the differing locations (lenses) from each edit are examples for trans
6: guard $\leftarrow$ PBESynth(DSL_G, GuardSpec)
7: trans $\leftarrow$ PBESynth(DSL_T, TransSpec)
8: **if** guard = $\bot \wedge$ trans = $\bot$ **then return** $\bot$
9: **if** Score(guard) < $\text{threshold}_g \vee$ Score(trans) < threshold_t **then return** $\bot$
10: **return** (guard, trans)

The problem solved by the oracle in the procedure OracleBasedExplanationGenerator can be stated as: given a set of edits $\{(v_{b_1}, v_{a_1}), (v_{b_2}, v_{a_2}), \ldots, (v_{b_n}, v_{a_n})\}$, find a pair of programs (guard, trans) such that there exist lenses $\ell_1, \ldots, \ell_n$ such that (a) $\ell_j \in \text{guard}(v_{b_j})$, and (b) $\ell^{set}_j(v_{b_j}, \text{trans}(\ell^{get}_j(v_{b_j}))) = v_{a_j}$ for $j = 1, 2, \ldots, n$.

Algorithm 2 presents a high level procedure that implements the oracle given a set of edits. The major components of the algorithm are as follows:

(a) *Diff computation.* Given an edit (v_b, v_a), the procedure DiffLoc computes the location (lens) whose content differs between v_b and v_a. In the canonical domain of ASTs, we use a recursive algorithm to find the smallest sub-tree that contains all the differences between the two ASTs. For each edit (v_{b_i}, v_{a_i}), let ℓ^i be the computed differing location.
(b) *Guard synthesis.* Given the input-output examples $\{v_{b_1} \mapsto \ell^1, \ldots, v_{b_n} \mapsto \ell^n\}$, we use PBE to synthesize a program guard such that $\ell^i \in \text{guard}(v_{b_i})$ for all i.
(c) *Transformation synthesis.* Given the input-output examples $\{\ell^1_{get}(v_{b_1}) \mapsto \ell^1_{get}(v_{a_1}), \ldots, \ell^n_{get}(v_{a_n}) \mapsto \ell^n_{get}(v_{a_n})\}$, find a program trans such that $\text{trans}(\ell^i_{get}(v_{b_i})) = \ell^i_{get}(v_{a_j})$.

The above two PBE problems can be solved independently. For each PBE problem, we can design a separate domain-specific language (DSL) based on the structure of the document and the kind of edits of interest. In our canonical application where documents are ASTs, we can use Refazer [Rolim et al. 2017] to synthesize both guard and trans since the DSL used by Refazer can express both components.

Ranking and the Bias-Variance Tradeoff. In Algorithm 2, the expressivity of DSL_G and DSL_T along with the thresholds threshold_g and threshold_t control the universe $\mathbb{P}$ of possible programs for generating explanations. Given a sufficiently expressive universe of programs any finite set of edits can be explained with a single program (that uses a top-level "switch" statement to separate each edit into a case of its own). In this case, the optimal explanation generation problem has a trivial solution, and the size of the optimal explanation is just 1. However, this is not very useful because that one program would not be very predictive of a future edit. A good explanation must therefore (a) generalize, without overgeneralizing, and (b) determine the appropriate context under which generalized edits are applicable. This is exactly the bias-variance trade-off seen in the context of machine learning; with the more and less expressive $\mathbb{P}$'s playing the role of over-fitting and under-fitting models, respectively.

Fortunately, the problem of capturing a user's intent by learning a program that performs a *controlled generalization* from concrete demonstrations (via input-output examples) is extensively studied in the PBE field [Polozov and Gulwani 2015; Singh and Gulwani 2012; Yaghmazadeh et al. 2018], mainly in the form of ranking functions. Informally, highly ranked programs are more likely to be the right generalizations. Hence, we leverage these ranking functions by using thresholds (threshold_g and threshold_t) to precisely control the set of programs $\mathbb{P}$ available to Blue-Pencil.

5.2 Reducing the size of the DAM

While constructing the DAM, the OracleBasedExplanationGenerator procedure considers all possible subsets of (transitive) edits. This is expensive and infeasible to do in real time. In this section, we describe multiple heuristic approaches for reducing the size of the DAM and for generating only promising subsets of edits; that is, subsets of edits that are likely to have a shared explanation.

Debouncing and Transient Edits. Common editing activities such as typing generates a large number of document versions. For example, a user typing the identifier `IsValid` is likely to generate a history with versions for each of the partial names `I`, `Is`, `IsV`, and so on. To avoid this and other related problems related to transient edits, we use a debounce strategy. Informally, we record all the versions V_{new} produced in a short time period called the *debounce duration* (500ms in our experiments). Now, all these vertices are incrementally added to the DAM using the procedure IncrementalVerticesAddition from Algorithm 3. For each new version v_{k+i}, this procedure checks if the version is transient in the context of the previous and next versions. Informally, a version is considered transient if the location of the next edit is the same as the location of the current edit, i.e., $\text{DiffLoc}(\text{prev}, v_{k+i}) = \text{DiffLoc}(v_{k+i}, v_{k+i+1})$. If the version is not transient, it is added to the DAM. Then, the procedure incrementally adds all possible edges from previously existing versions to the newly added version using the IncrementalEdgeAddition procedure (see below).

Promising Edit Subsets. The second heuristic avoids calling the oracle on all subsets of edges, and instead only calls the oracle on subsets of edges that are likely to produce a common edit explanation program. Our approach is based on defining a map, `embed`, from document space to n-dimensional Euclidean space $\mathbb{R}^n$. This map has the property that two documents that are similar to each other (say, two versions of the same document that differ by just a few edits) map to points that are close in the Euclidean space. In the case when the document is an abstract syntax tree of a program – as is the case in our canonical example – there is existing work that already defines such a map [Jiang et al. 2007], which we reuse in our work. However, unlike this previous work, we are not interested in code clone detection, but we want to detect "edit" clones. Consequently, we define an Euclidean space embedding of edits as

$$\text{embed}((v, v')) = \text{embed}(v') - \text{embed}(v)$$

Let $\text{KNearestNeighbors}(e, \text{edits}, K)$ return the K edits from the set edits that are closest to the given edit e in the Euclidean space. Given a new (transitive) edit (v_i, v_j), the IncrementalEdgeAddition procedure works by finding a promising set of "similar" edits using KNearestNeighbors. Non-overlapping subsets of these similar edits are passed to the PBE oracle to generate a common explanation for these sets. Since the number K is a fixed small constant, the running time of the procedure is only polynomial, and not exponential, in the size of the DAM (V, E).

5.3 Greedy procedure for finding short explanations

Let (V, E) be the DAM where each node in V corresponds to a version of the document in the given history $v_0 v_1 \cdots v_k$, and each edge $(v_i, \mathsf{P}, v_j) \in E$ is labeled with a program P that explains the edit

Algorithm 3 Incremental construction of the DAM when a set of new vertices V_{new} are added.

```
1: function IncrementalVerticesAddition(V, E, V_new)
2:     Say V = {v_0, ..., v_k} and V_new = {v_{k+1}, ..., v_{k+k'}}
3:     prev ← v_k
4:     for each v_{k+i} ∈ V_new do
5:         if ¬IsTransient(prev, v_{k+i}, v_{k+i+1}) then
6:             prev ← v_{k+i}, V ← V ∪ {v_{k+i}}
7:             for v ∈ V \ v_{k+i} do
8:                 E ← E ∪ IncrementalEdgeAddition(V, E, (v, v_{k+i}))
9:     return (V, E)
10: function IncrementalEdgeAddition(V, E, (v_i, v_j))
11:     newEdges ← ∅
12:     oldEdits ← {(v_a, v_b) | 0 ≤ a < b ≤ i}
13:     similarOldEdits ← KNearestNeighbors((v_i, v_j), oldEdits, K)
14:     for each nonoverlapping subset of similarOldEdits that contains (v_i, v_j) do
15:         if PBEOracle(subset) = P ≠ ⊥ then
16:             newEdges ← newEdges ∪ {(v, P, v') | (v, v') ∈ subset}
17:     return newEdges
```

Algorithm 4 Greedy procedure for finding a short explanation

```
1: function GreedyExplanation(V, E)
2:     explanation ← ∅
3:     while E ≠ ∅ do
4:         P* ← argmax_P len(Span(E(P)))
5:         E ← E \ {e ∈ E | Overlap(e, E(P*)) = True}
6:         E ← E \ {(v_i, P, v_j) ∈ E | P ≠ ⊥_i ∧ |E(P)| = 1}
7:         explanation ← explanation ∪ {P*}
8:     return explanation
```

$(v_i, v_j), i < j$. In this section, we present a greedy approach to find a path from the node v_0 to the node v_k that minimizes the number of distinct programs P used as edge labels in the path.

When we add edges labeled by P to the DAM, for any fixed P, these edges are all mutually non-overlapping. Let $E(\mathsf{P})$ denote the edges in E labeled with P; that is, $E(\mathsf{P}) = \{(v, \mathsf{P}, v') \in E\}$. For simplicity, assume that, for any fixed P, the edges in $E(\mathsf{P})$ are always non-overlapping. This is a reasonable assumption since it is unlikely that two overlapping edits will share the same edit explanation program P. Define the span, Span, of an edge (v_i, P, v_j) as the interval $[i, j]$, and define the span of non-overlapping edges as the union of the spans of each edge; that is,

$$\mathsf{Span}((v_i, \mathsf{P}, v_j)) = [i, j], \qquad \mathsf{Span}(E') = \bigcup_{e \in E'} \mathsf{Span}(e)$$

The length $\mathtt{len}(\mathsf{Span}(E'))$ is defined as $\sum_{(v_i, \mathsf{P}, v_j) \in E'} (j - i)$.

Algorithm 4 shows pseudocode for the greedy approach for finding a path from v_0 to v_k. The set E of edges keeps decreasing as we add more programs P to the set explanation. The procedure greedily picks the program that covers most of the uncovered fine-grained edits. In practice, we find that the greedy procedure produces explanations of reasonable size.

Algorithm 5 On the fly suggestion generation when user creates version v_{k+1}

```
1:  function OnTheFlySuggestions(V, E)
2:      while True do
3:          V_new ← collect new document versions for debounce period
4:          (V, E) ← IncrementalVerticesAddition(V, E, V_new)
5:          explanation ← GreedyExplanation(V, E)
6:          suggestions ← ∅
7:          for every p = (guard, trans) in explanation that is not ⊥_i do
8:              new_suggestions ← {(ℓ, u) | ℓ ∈ guard(v_{k+k'}) ∧ u = trans(ℓ_get(v_{k+k'})) ∧ u ≠ ⊥}
9:              suggestions ← suggestions ∪ new_suggestions
10:         yield return suggestions
```

guard	::=	Select(*path*)	*transformation*	::=	Update(*ast*) | InsertChild(*ast*, *k*)
path	::=	Path($step_1$, …, $step_n$)		|	DeleteChildrenAt(*pos*, *pos*)
step	::=	Step(*kind*, *k*, *pred*)	*ast*	::=	*const* | Reference(*guard*, k)
pred	::=	Exists(*path*) | *pred* ∧ *guard* | ¬ *pred*	*const*	::=	Node(*kind*, *attributes*, ast_1, …, ast_n)

Fig. 2. The composite DSL over which guards and transformations are synthesized in the Blue-Pencil instantiation for code transformations.

5.4 Incremental procedure: Putting it all together

We finally have the heuristically optimized versions of the three components of the procedure `OracleBasedExplanationGenerator`. These heuristics combined give us both performance and ambiguity resolution. We can now put them together to obtain a procedure that can suggest future edits on a document.

The Procedure `OnTheFlySuggestions`, shown as Algorithm 5, maintains a representation of the DAM (V, E). As the user is editing, the procedure collects new document versions for the debounce period (500ms), and adds the new versions to the DAM using the IncrementalVerticesAddition procedure. Once we have updated the DAM, we use procedure `GreedyExplanation` to generate an explanation for the edit history $v_0 v_1 \cdots v_{k+k'}$ using the greedy approach. Finally, we run each of the programs found in the explanation on latest $v_{k+k'}$ to generate a number of suggestions. Each suggestion is of the form (ℓ, u), suggesting that the contents of the location ℓ should be replaced by the new contents u. Our user interface then shows a "Code Action lightbulb" [Microsoft 2019d] and a "warning squiggle" that suggests this edit. A screenshot of the Visual Studio extension for Blue-Pencil showing edit suggestions after the user modifies two locations is presented in Figure 9.

6 INSTANTIATIONS OF BLUE-PENCIL

The canonical instantiation of Blue-Pencil, that we have focused on throughout this paper is applicable to the domain of repetitive code transformations. We now describe the PBE components used in this instantiation. Further, we also include a brief discussion of our experience with building other instantiations of Blue-Pencil.

6.1 Blue-Pencil for code transformations

Figure 2 shows the domain-specific language (DSL) over which the guards and transformations are synthesized in the instantiation of Blue-Pencil for code edits. The DSL is similar to the one described in earlier work [Rolim et al. 2017]. Programs derived from the non-terminal named

	Document Class	DSL for Guards	DSL for Transformations	Location/View Type
1	Code/AST	Refazer	Refazer	Sub-trees of AST
2	Markdown	Refazer	Refazer + FlashFill	Sub-trees of AST and text values from leaf nodes
3	Spreadsheets	FlashProfile	FlashFill	Text contents of table cells

Table 1. Characteristic features of other instantiations of Blue-Pencil.

guard in Figure 2 are used as guards in our implementation. And programs rooted at the non-terminal named *transformation* are used as transformations. Effectively, the lenses select and update (sub-trees of) ASTs, while the transformations are performed over ASTs as well.

Guard expressions. A guard is represented by the Select operator, which selects every node n in the input AST x that satisfy a guard represented as a path p. To represent paths, we use a language inspired by XPath language [Refsnes Data 2019], which allows us to select an node n in x by following a sequence of steps in the AST. The Path operator has a sequence of steps step_{1-n}, where the first step searches in $\text{children}^*(x)$ (denoted as "//" in XPath), and the following steps searches only in $\text{children}(t)$, where t is the current AST (denoted as "/" in XPath). Each step selects the k^{th} occurrence (or all occurrence if k is not specified) of an AST of kind *kind* where the predicate *pred* holds. Predicates check the (non-)existence of elements in the AST.

Transformation expressions. A transformation is either an Update, an Insert or a Delete operation on the given AST node. The sub-tree *ast* to be inserted or updated is built recursively out of either constants or references to parts of the original node (selected by path specifiers).

6.2 Other instantiations of Blue-Pencil

We have alluded to the parameterization of Blue-Pencil several times. As proof-of-concept, we have built implementations that automate repetitive edits for other document types to verify that Blue-Pencil can indeed be adapted to other domains with minimal effort. Table 1 lists the other instantiations, and describes the nature of the PBE engines used in them.

The first row of Table 1 shows the characteristics of Blue-Pencil for code edits, which forms the main focus of this work. We briefly describe the other instantiations now. We do not conduct an extensive evaluation of the instantiations of Blue-Pencil for markdown and for spreadsheets: they are proof-of-concept implementations that serve to demonstrate that Blue-Pencil's parameterization permits application to a variety of domains. Our experience demonstrates that Blue-Pencil can be adapted to automate repetitive edits in domains other than code editing with a reasonable amount of effort.

Blue-Pencil for Markdown documents. Markdown is a simple language for writing structured documents [CommonMark 2019]. It is designed to be readable in source form, while also supporting compilation into HTML or other layout languages. We implemented a proof-of-concept of Blue-Pencil instantiated to automate repetitive edits in Markdown documents. The total effort required was about four person-days of work, which includes the time spent in adapting the UI (implemented as a Visual Studio Code plugin) for Markdown. However, we did not implement the optimization based on KNearestNeighbours.

The prototype supports both structural transformations in Markdown (which are implemented as transformations on trees), as well as transformations on strings which occur in the leaf nodes of the Markdown AST. We instantiated Blue-Pencil with the guard DSL from Refazer [Rolim et al.

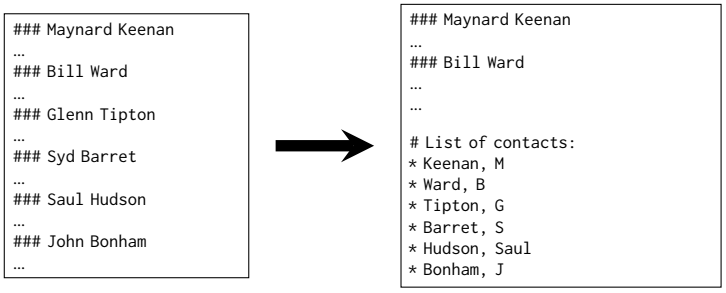

Fig. 3. An example of a repetitive edit in Markdown.

2017]. The transformation DSL was a combination of the DSL from Refazer (for the structural/tree transformations), and the DSL from FlashFill [Gulwani 2011] (for string transformations).

The specific scenarios that we tested are listed below. The scenarios included non-repetitive edits that were performed during the editing session.

- Converting headings of level n to level k, $n \neq k$.
- Phone number reformatting: reformat a phone number in a paragraph written as ##########, i.e., as just 10 decimal digits into the format ###-###-####.
- Email reformatting: extract just the user name part of emails from a list of emails.
- Composite formatting and structural edits: See Figure 3. The original document shown on the left in Figure 3 contains names of celebrities as level 3 headings, followed by a brief biography (indicated with ellipses). The task is to collate the names of all of these celebrities into a list, while simultaneously reformatting their names as shown on the left in Figure 3. Our prototype was able to learn the intended transformation after the user manually populated the first two entries of the list, and automatically populated the remaining four entries of the list.

Blue-Pencil for spreadsheets. We implemented Blue-Pencil for spreadsheets as a Visual Studio Code plugin– extending the existing plugin to spreadsheets (represented as CSV files) took 3 person hours of work – however, we did not implement the optimization based on KNearestNeighbours. The guards for this implementation consist of a column number k and a regular expression r; a guard given by k and r selects exactly those cells from column k whose contents match the regular expression r. The regular expressions r in the guards were synthesized by a variant of the technique FlashProfile [Padhi et al. 2018]. Given a set of strings, FlashProfile can generate a set of regular expressions that together matches strings similar to input strings; we modified this technique to produce a single regular expression. We use FlashFill for the transformation DSL.

As compared to the widely deployed FlashFill PBE system for spreadsheets, Blue-Pencil allows:

(a) In-place modification of cell values to provide examples – no requirement of creating a new column to enter into PBE mode,
(b) Noise and non-repetitive edits – Blue-Pencil filters out non-repetitive modifications, and
(c) Better support for selective and conditional transformations through FlashProfile generated regular expressions – FlashFill by itself generates incorrect outputs in some cases due to applying the learned transformation on cells that are dissimilar to the inputs in the examples.

Blue-Pencil was able to synthesize suggestions to automate the edits listed below. As before, the editing session included noise in the form of non-repetitive edits

(1) Replacing middle names with middle initials in a list of names (e.g. `'Arthur Charles Clarke'` to `'Arthur C. Clarke'`). The list contained names both with and without middle names—the guard selected only the names with middle names (see Figure 4).

Author	DOB	Language
Arthur Charles Clarke	16 December 1917	English
Surender Mohan Pathak	19-Feb-40	hindi
Alexandre Dumas	24 July 1802	French
Jorge Luis Borges	24 August 1899	Spanish
Haruki Murakami	12-Jan-49	Japanese
Barbara Cartland	9-Jul-01	English
…	…	…

Before

Author	DOB	Language
Arthur C. Clarke	1917	English
Surender M. Pathak	1940	Hindi
Alexandre Dumas	1802	French
Jorge L. Borges	1899	Spanish
Haruki Murakami	1949	Japanese
Barbara Cartland	1901	English
…	…	…

After

Orange, blue, and red edits refer to repetitive user edits, Blue-Pencil suggestions, and non-repetitive user edits, respectively.

Fig. 4. Blue-Pencil on spreadsheets.

(2) Extracting the year from a column of differently formatted dates (e.g., `'11 November 1958'` to `'1958'` and `'14-03-77'` to `'1977'`). Here, multiple programs were learned (one for each date format) with the learned guards (regexes) exactly matching dates of different formats.
(3) Reformatting negative numbers (from a column of both positive and negative numbers) from standard to accounting format (e.g., `'-5.27'` to `'(5.27)'`). Here, the guard was to select cells that are floating point numbers starting with a minus sign.

In each of these cases, FlashFill by itself (without guards) produces the wrong outputs due to applying the transformation on cells which do not match the intended guard.

7 EVALUATION

We present experiments that evaluate Blue-Pencil with respect to its effectiveness and efficiency. In particular, we instantiate Blue-Pencil for code transformations (C# and SQL), as presented in Section 6.1, and we aim to answer the following research questions:

RQ1 *How accurate are the suggestions produced by Blue-Pencil in scenarios of repetitive and non-repetitive edits?*
Producing incorrect suggestions while the user is editing a document can be a big barrier for the adoption of such systems. We measure the number of false positives (incorrect suggestions), false negatives (missing suggestions), and true positives (correct suggestions) produced by Blue-Pencil.

RQ2 *How fast can Blue-Pencil generate suggestions?*
Blue-Pencil needs to quickly generate suggestions, otherwise the user will edit the locations before suggestions are produced. We measure the time Blue-Pencil takes to learn suggestions.

RQ3 *Can Blue-Pencil help programmers perform repetitive edits?*
Ultimately, Blue-Pencil is intended to help programmers do their jobs better. We perform a field study (Section 7.6) to understand how Blue-Pencil performs when used in everyday development.

7.1 Benchmark Suite

We recorded 37 document editing sessions in the context of software development in two languages (C# and SQL). These sessions came from three different sources:

Repositories We mined 5 scenarios of repetitive edits on C# programs from 3 public GitHub repositories and 6 scenarios of repetitive edits on SQL programs from 3 private repositories. Since the repository does not provide the fine-grained edits performed by the programmer but instead provides the diff between two commits, we recorded two professional software engineers performing these edits. In total, we recorded (5 + 6) * 2 = 22 editing sessions. For each scenario, the software engineer was presented with the original code, a description of the task that should be performed containing one example of the repetitive edit, and the number of locations to edit.

Authors' scenarios We collected 10 sessions containing repetitive edits during the development of Blue-Pencil. These sessions were recorded from one of the authors of the paper, and they are noisy, i.e., they contain non-repetitive edits performed before, between, and after the repetitive edits.

Non-repetitive edits We recorded 15 min of non-repetitive edits divided in 5 small sessions.

In total, our benchmark suite consists of 22 + 10 + 5 = 37 code editing sessions, which contain 11 + 10 = 21 distinct repetitive edit scenarios. Let us first determine what to expect from Blue-Pencil on the 21 scenarios. Note that Blue-Pencil is parameterized by the PBE engines, and since it was not our goal to evaluate the PBE engines, we picked the 21 scenarios so that they would be detected as repetitive in theory. We first determined, by hand, the number of examples the PBE engine would definitely need to synthesize a program that explains all the repetitive edits. Figure 5 shows the total number of repetitive edits in each scenario and the minimum number of examples PBE would need for that scenario. For example, three examples are necessary to synthesize the correct program in Scenario 0. So no two examples, when given to the PBE engine, would synthesize the correct program. On the other hand, while three carefully chosen examples would synthesize the correct program, more examples could be required if the wrong ones were provided. We found that, on average, Blue-Pencil should be able to automate 64% of each benchmark's edits (86 out of 135 total edits). The *granularity* of a repetitive edit instance is the number of fine-grained edits that instance is comprised of. The average granularity of the repetitive edit instances was found to be 3.4 and the largest granularity was 17. The average size of the number of modified nodes in the AST was 32 and the largest size was 99. The benchmark suite contains a variety of edits, from small edits that need to be applied to many locations to large edits that only need to be applied to a few locations.

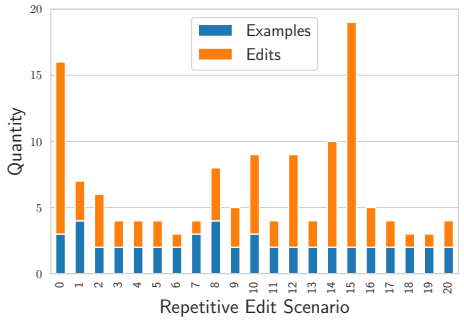

Fig. 5. The total number of repetitive edits in each scenario and the minimum number of examples needed by the PBE engine to learn the correct suggestions. The orange color represents the edits that Blue-Pencil should suggest in theory.

7.2 Experimental Setup

For each recorded session, we simulate the use of Blue-Pencil by running it across the history of document versions. To simulate a realistic environment, where Blue-Pencil gets a stream of document versions, a suggestion suite is incrementally synthesized for each version. For sessions with repetitive edits, we run Blue-Pencil until it has seen enough repetitive edits for the PBE system to learn the correct program (as defined by Figure 5), assuming perfect edit sets as examples. For instance, if the scenario has 20 repetitive edits and the underlying PBE system requires two

Table 2. Blue-Pencil configurations used in the experiment.

Configuration	Threshold	Transient	KNN	Explanation
$BP_{\text{-threshold}}$	✗	✓	✓	✓
$BP_{\text{-transient}}$	✓	✗	✓	✓
$BP_{\text{-KNN}}$	✓	✓	✗	✓
$BP_{\text{-explanation}}$	✓	✓	✓	✗
BP	✓	✓	✓	✓

Table 3. Summary of the results for RQ1. Suggestions = number of suggestions produced by Blue-Pencil; False Positives = incorrect suggestions; False positives = missing suggestions.

Configuration	Suggestions	False positives	False negatives	Precision	Recall
$BP_{\text{-threshold}}$	998	815	0	0.18	1.00
$BP_{\text{-transient}}$	313	134	0	0.57	1.00
$BP_{\text{-KNN}}$	206	23	0	0.89	1.00
$BP_{\text{-explanation}}$	234	51	0	0.78	1.00
BP	206	23	0	0.89	1.00

examples to learn the correct program, we incrementally run Blue-Pencil until the end of the second example. For sessions that do not have repetitive edits, we run Blue-Pencil until the last version of the document.

We measure the time to run Blue-Pencil at each document version, and calculate the precision and recall of all the suggestions produced by Blue-Pencil over a session. To evaluate the impact of each component of Blue-Pencil on its correctness and performance, we run it with the 5 different configurations shown in Table 2. *BP* uses all components proposed in Section 5: (i) elimination of low-scored programs produced by the DSL (Threshold); (ii) elimination of transient versions; (iii) KNearestNeighbors-based search of edit sets (KNN); and (iv) greedy search of short explanations (Explanation). We set K to 5 and the threshold for the minimum score of the programs to 400. These values were empirically determined during the development of Blue-Pencil but before collecting the benchmark from the external software engineers. Each other configuration removes its corresponding component: (a) $BP_{\text{-threshold}}$ uses all programs returned by the PBE system, (b) $BP_{\text{-transient}}$ does not eliminate transient versions, (c) $BP_{\text{-KNN}}$ search over all non-overlapping edit sets, and (d) $BP_{\text{-explanation}}$ returns all possible explanations.

All our experiments were run on Intel Xeon E5-1620 v4 CPU running at 3.5 GHz with 32 GB RAM running 64 bit Windows 10 Enterprise. The hardware is consistent with professional software developer workstations.

7.3 Results

Table 3 and Table 4 summarize the results of our experiments. Table 3 presents the number of synthesized suggestions, false positives, false negatives, precision, and recall. Table 4 presents the time to invoke Blue-Pencil and number of synthesis invocations that Blue-Pencil did to underlying the PBE system.

> The precision and the recall of Blue-Pencil (*BP*) were 0.89 and 1.0, respectively. The time to generate suggestions was 199 ms on average. These results suggest that Blue-Pencil is effective and efficient enough to produce edit suggestions on the fly.

Table 4. Summary of the results for RQ2; time = time to invoke Blue-Pencil; Syn. inv. = number of synthesis invocations; Suc. syn. inv = number of synthesis invocations that successfully return a program.

Configuration	Mean time (ms)	Std. dev. time (ms)	Syn. inv.	Suc. syn. inv.
$BP_{\text{-threshold}}$	299.85	482.10	2792	187
$BP_{\text{-transient}}$	952.15	1586.48	9725	264
$BP_{\text{-KNN}}$	500.90	1863.43	8339	198
$BP_{\text{-explanation}}$	222.43	406.13	2791	121
BP	198.91	348.88	2791	121

Table 5. Classification of false positives. Transient false positives are incorrect suggestions that were generated while the user is editing the repetitive edit but removed when the user finished editing the code. Permanent false positives are false positives that were still presented after the user finish editing the code.

Configuration	Transient (%)	Permanent (%)
$BP_{\text{-threshold}}$	61 (0.07)	754 (0.93)
$BP_{\text{-transient}}$	117 (0.87)	17 (0.13)
$BP_{\text{-KNN}}$	23 (1.00)	0 (0.00)
$BP_{\text{-explanation}}$	2 (0.04)	49 (0.96)
BP	23 (1.00)	0 (0.00)

7.4 Discussion

Causes and Prevention of False Positives. With all optimizations included, Blue-Pencil generated 43 false positives, which occurred across five editing sessions. Table 5 classifies the false positives into two categories: transients and permanents. The former represents false positives that were suggested while the user performed the edit, but were no longer suggested after the user completed that change. These suggestions are usually small and represent some component of the full repetitive edit. Filtering programs that have a low ranking score is crucial to avoid these false positives. When we remove this filter ($BP_{\text{-transient}}$), Blue-Pencil produced 134 false positives, from which 87% were transient false positives.

Permanent false positives are incorrect suggestions that are still presented after the user finished typing the repetitive edit. These false positives can impact user experience more, since they persist in the document and make it harder for the user to decide which suggestions should be applied. Finding the best explanation for the history is important to avoiding this type of false positive. When we remove the component that selects the best explanation (as in $BP_{\text{-explanation}}$), 97% of the false positives produced were permanent. This indicates that our definition of optimal explanations being those with a minimal number of programs helps capture user intent, for it prevents false positives while still avoiding false negatives.

To illustrate this scenario, consider the repetitive edit showed in Figure 6. In this session, the programmer moved the logic in the red lines to an existing method, avoiding the duplication. To do so, the programmer quickly deleted statement by statement in both locations. There was still one more location to apply this edit. The correct suggestion is shown in Figure 7a. Two of permanent false positives produced by $BP_{\text{-explanation}}$ are shown in Figures 7b and 7c. Presenting all these possible edits to the user can be confusing and reduce the confidence of the user in the system.

```
   //...
    if (newNode != null)
    {
-         var matchScore = rule.MatchProgram.Score;
-         var editScore = rule.EditProgram.Score;
-         if (matchScore > matchThreshold && editScore > editThreshold)
-         {
              return newNode;
-         }
     }
   //...
    if (newNode != null)
    {
-         var matchScore = rule.MatchProgram.Score;
-         var editScore = rule.EditProgram.Score;
-         if (matchScore > matchThreshold && editScore > editThreshold)
-         {
              node.Children[i] = newNode;
              newNode.Parent = node;
              break;
-         }
     }
   //...
```

Fig. 6. Minified version of the repetitive edit performed in the recorded session 7. The programmer moved the logic duplicated in these two locations to a different part of the code removing the duplication.

Resiliency to False Negatives. In our experiments, the recall for all configurations was 1.00, which suggested that the two approaches proposed to reduce false positives (ranking programs and explanations) and the approach proposed to reduce the search space (KNN-based search) do not have a negative impact on recall.

Impact of KNN Clustering. $BP_{\text{-KNN}}$ generated the same suggestions and false positives of *BP*, which suggests that using a KNN-based search (instead of searching over all edit sets) does not have a negative impact on the precision.

The KNN-based search had a significant impact on the performance of Blue-Pencil. When this component is removed (as in $BP_{\text{-KNN}}$), the average time increased to 500.90 ms. Additionally, there was a much higher variation (1863.43 ms). This difference can be better visualized in Figure 8, where we plot the time for every invocation of Blue-Pencil with and without the KNN-search. We can observe that some of the invocations of $BP_{\text{-KNN}}$ took almost 30 seconds to be performed, which can be a barrier to synthesize suggestions on the fly, *BP* invocation time was much more concentrated on the left part of the plot. We can also observe that removing transient versions of the history not only reduces false positives but has a significant positive impact on the performance since it makes the history smaller. Table 4 shows that the number of invocations to the PBE system was much higher in $BP_{\text{-KNN}}$ and $BP_{\text{-KNN}}$ compared to the invocations in *BP*, while the number of successful invocations were much closer, which suggests that without the KNN-based search, the system spends much more time trying to synthesize programs from sets of edits that are inconsistent.

7.5 Limitations

Our benchmark is not representative of all possible repetitive edits that users may perform. To reduce this threat, we selected repetitive edits performed in two different languages, and that vary

```
-   var matchScore = rule.MatchProgram.Score;
-   var editScore = rule.EditProgram.Score;
-   if (matchScore > matchThreshold && editScore > editThreshold)
-   {
        suggestions.Add(new Suggestion(location,transformation));
-   }
```

(a) Correct suggestion.

```
-   var matchScore = rule.MatchProgram.Score;
    var editScore = rule.EditProgram.Score;
    if (matchScore > matchThreshold && editScore > editThreshold)
    {
        suggestions.Add(new Suggestion(location,transformation));
    }
```

(b) Incorrect suggestion that deletes just one statement.

```
-   var matchScore = rule.MatchProgram.Score;
-   var editScore = rule.EditProgram.Score;
-   if (matchScore > matchThreshold && editScore > editThreshold)
    {
        suggestions.Add(new Suggestion(location,transformation));
    }
```

(c) Incorrect suggestion that deletes 3 statements, including the if statement, but does not delete the inner braces.

Fig. 7. Example of suggestions produced by $BP_{\text{-explanation}}$ after the edits showed in Figure 6.

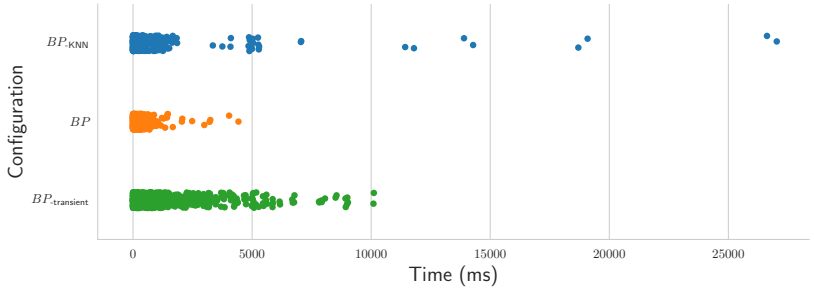

Fig. 8. Performance comparison between BP and $BP_{\text{-KNN}}$.

in granularity, size, and number of locations. Additionally, we included sessions that contained only non-repetitive edits.

The traces of the edits collected from software repositories were replicated to create a history of document versions. To reduce bias during the replication, we asked for external software engineers to apply these edits. However, the way they performed the edit may not represent the exact way the edit was performed originally.

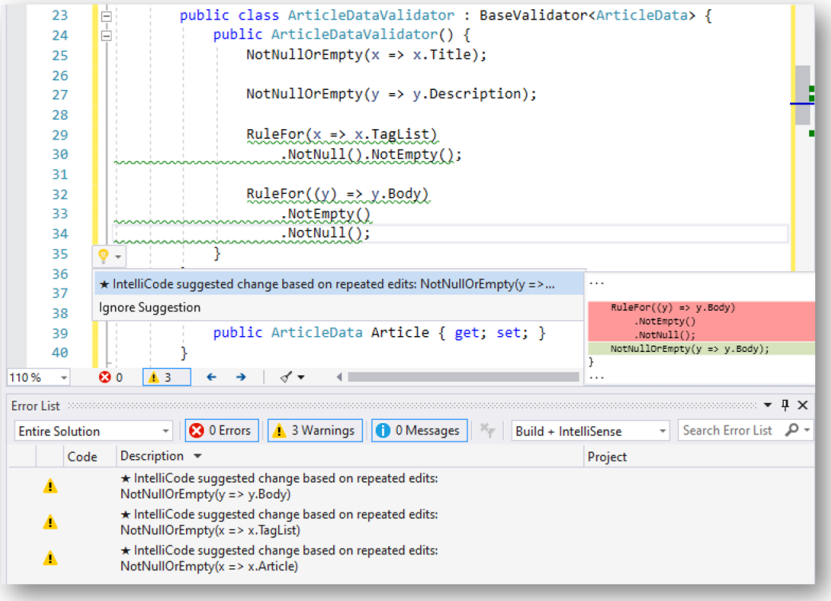

Fig. 9. Blue-Pencil implemented as the "*refactorings*" detection feature in the IntelliCode extension.

7.6 Field Study

In addition to the in-vitro experiments above, we conducted a field study to investigate Blue-Pencil's effectiveness in a less controlled environment and collect qualitative feedback. In collaboration with the Microsoft IntelliCode[3] team, we implemented the proposed algorithm as the "*refactorings*" feature in the IntelliCode extension for C#.

Figure 9 illustrates the extension usage. In this scenario, a programmer performed two repetitive edits to remove duplicated code by calling the method `NotNullOrEmpty`. The extension recognized the pattern and immediately produces three other code edit suggestions. All suggestions are shown as warnings in the "Error list" panel. Additionally, the green[4] squiggles in the code and the green boxes in the scroll bar help the programmer identify the locations where similar edits can be performed. By clicking on the Light Bulb Action icon next to the squiggle, the programmer can see a preview of the suggestion, apply the suggestion, or ignore it. To collect feedback, we created a bug report channel and conducted informal interviews with the programmers.

We shipped the extension to an internal group of programmers at Microsoft. Over a period of three months, we had around 25 programmers using the tool each week. Since many programmers code in different languages, they were not using the tool all the time.

In our study, programmers applied suggestions produced by Blue-Pencil, including the suggestions shown in Figure 1. The qualitative feedback provided by them was encouraging. For instance, one programmer mentioned: *"I did a refactor* [sic] *in a method, changed the output to an enumerable and had to update the tests. The extension caught the pattern pretty quickly"*. Next we describe three aspects of the technology that were improved during the study.

[3] https://visualstudio.microsoft.com/services/intellicode/

[4] We used green squiggles instead of blue ones to be consistent with other suggestions produced by Visual Studio.

Usability. During the first month of the study, programmers reported usability barriers for the use of the suggestions. They raised two main issues. The first one was related to discoverability. Participants were not seeing the suggestions produced by Blue-Pencil. Initially, lines in the code that contain suggestions were marked just with the symbol ... in the beginning of the line, which is the default option in Visual Studio for code actions. We improved the discoverability of the tool by adding the green squiggle and listing the suggestions as warnings. The second issue was related to the lack of the preview feature, which made it difficult to review the changes. We added this feature later. Finally, programmers asked for a feature that allows them to apply all suggestions at once. We plan to include this feature in the future.

Accuracy. Programmers reported false positives and false negatives produced by the tool. Although we collected more than 30 bugs related to these issues, only one programmer reported annoyance with the false positives. Most of the bugs arose from two sources: short-comings of the underlying PBE synthesizer (Refazer) and threshold for the scores of the programs (Section 5.1). Originally, guard expressions produced by Refazer considered just the AST related to the location being changed and the parent node of this AST. We observed that this context was not enough in many cases. Specifically, when the changed sub-tree of the AST was small, the guard expression was usually overly general. For instance, changing `x` to `(int) Math.Sqrt(x)` in an argument list could be overgeneralized to do this edit to every variable in all argument lists in the code. We improved the guards to consider more context such as method call information. Improving Refazer and fine-tuning the ranking scores solved most of the issues. We plan to add more features to Refazer, such as typing information, to improve the context more and eliminate the other false positives. Additionally, there were false positives related to edits that were repetitive but that required more semantic information about the task being performed to decide whether the suggestion should be generated. For instance, changing the types of two variables from `List<int>` to `List<string>` during a programming session is a repetitive edit but it does not mean that the programmer wants to do it for all variables of type `List<int>` in the code. We added a button to allow programmers to ignore suggestions.

Performance. Programmers have not reported any issue related to CPU usage while running Blue-Pencil in the background and the tool was fast enough to produce suggestions in real time. However, there were concerns related to the memory usage of the tool. Since Blue-Pencil stores multiple versions of the code, on large files (> 1000 LOC), the amount of memory used to produce the ASTs corresponding to each version of the file was prohibitive. We performed multiple optimizations to our data structures in order to keep the memory used by Blue-Pencil under 50 megabytes per session.

To summarize, the feedback was useful to confirm the viability of Blue-Pencil in terms of accuracy and performance in a real programming environment, and helped us make the tool more robust. Given the success of the initial field study, as the next step, we plan to run a more formal and involved quantitative study with more programmers using the *IntelliCode refactorings* feature of IntelliCode that is based on Blue-Pencil.

8 RELATED WORK

Inductive Program Synthesis. Inductive program synthesis has been an active research area for decades [Cypher 1993; Lieberman 2001]. LAPIS is a text editor that takes a set of positive and negative examples (i.e., text selections) from a file, learns a pattern and highlights similar matches in the file [Miller and Myers 2002]. FlashFill learns syntactic string transformation from the input/output examples in spreadsheets [Gulwani 2011]. FlashExtract extracts hierarchical data by examples [Le and Gulwani 2014]. Parsify synthesizes parsers from input/output examples [Leung

et al. 2015], while λ^2 targets data structure transformation [Feser et al. 2015]. Recently, FlashMeta (aka PROSE) reduces the effort in developing these domain-specific synthesizers by reusing the synthesis algorithms [Polozov and Gulwani 2015]. Our work shows its applicability to a different domain - learning program transformations.

Program Transformation from Examples. Andersen et al. proposed techniques to help update Linux device drivers when Linux internal libraries evolve [Andersen and Lawall 2008; Andersen et al. 2012]. Their approach generates generic patches from a set of files and their updated versions, and applies these patches on other files. Because they only focus on API usage changes, their underlying DSL is limited. Blue-Pencil has a more expressive DSL, thus can support a wider range of transformations.

Meng et al. [2011] introduced Sydit, a program transformation tool that generates a context-aware edit script from a single example. Sydit characterizes the code changes as a sequence of *edits*, from which it abstracts edit positions and identifiers to obtain the script. Subsequently, the authors extended the work in LASE to avoid the need to provide the edit locations and allow multiple examples [Meng et al. 2013]. Rolim et al. [2017] leveraged state-of-the-art PBE methodology to develop automatic transformation tool Refazer. While Blue-Pencil shares some similarities with these work (e.g., both require synthesizing the guards and edits, both accept multiple examples), it is quite different. Unlike previous work, Blue-Pencil continuously monitors the code changes and suggests the edit on the fly, which requires a special mechanism to effectively store, locate, and synthesize the transformations.

Edit Suggestion and Completion. Martin et al. introduced PQL, a declarative language, to allow programmers search for code that violates a design rule [Martin et al. 2005]. Unlike Blue-Pencil, it only suggests the violating locations. LibSync learns API usage adaptation patterns from existing migrated code [Nguyen et al. 2010]. LibSync only recommends the locations and the transforming operations. Programmers still have to write the changes manually based on these recommendations. In contrast, Blue-Pencil can generate fully executable transformations. WitchDoctor monitors programmer edits in the IDEs and completes refactorings that users begin performing refactorings themselves [Foster et al. 2012], but requires a built-in set of previous suggestion programs.

Big Code is a recent trend that leverages a large existing codebase for repetitive edit tasks. Nguyen et al. presented an API recommendation system based on statistical learning from fine-grained code changes and their context. Raychev et al. developed a code completion tool based on statistical language models [Raychev et al. 2014]. Recently, machine learning methods have been used to learn representations of edits on code and text documents, which a "neural editor" can then apply elsewhere [Yin et al. 2019]. Data mining techniques have been applied to large sequences of fine-grained edit data to find common, but previously unknown refactorings [Negara et al. 2014]. While these approaches require training from a large number of examples, Blue-Pencil can learn transformations from just a few examples.

Software Refactoring. Software refactoring is the process of changing the software system in a way that it preserves the external behavior yet improves the internal structure [Opdyke 1992]. In their classic survey, Mens and Tourwe [2004] described several aspects of software refactoring: the types of refactoring, the techniques to perform refactoring, the artifacts being refactored, the issues during implementation of refactoring tools, and the effect of refactoring. IDEs such as Visual Studio [Microsoft 2019e], Eclipse [Eclipse Foundation 2019], and IntelliJ [JetBrains 2019a] have some simple, yet popular, refactoring tasks built-in. These tasks (e.g., renaming variables or functions, changing function signatures) usually preserve the semantics and are implemented based on fixed templates. In contrast, Blue-Pencil supports transformations that may not be semantic-preserving.

Its transformations are *context-sensitive* – they may not make sense when applied to codebases other than the one it was learned from. Such transformations should not be packaged within an IDE, but are still useful when applied in right context. A recent field study by Kim et al. shows that in practice refactoring does not necessarily have to be semantic-preserving [Kim et al. 2012]. Blue-Pencil does not limit itself in refactoring; it can be used for other software evolution tasks.

Work on ad-hoc refactorings (also called "refactorings without names") [Steimann and von Pilgrim 2012] aims to generate specialized, context-specific refactorings not intended for general reuse. Unlike Blue-Pencil, their tool requires the programmer to enter a special mode in which to demonstrate a refactoring, after which the system suggests a number of candidate generalizations. Because the suggested changes must preserve program semantics, while those suggested by Blue-Pencil need not, we view their contribution as complementary to ours. Additionally, we speculate that instantiating Blue-Pencil with such an ad-hoc refactoring generator would enable it to suggest semantics preserving refactorings as well.

9 CONCLUSION

In this work, we introduce Blue-Pencil, a system to learn on the fly document edit suggestions by watching user interactions. We formalize the edit suggestion problem, and develop a parameterized algorithm to quickly find edit suggestions in different domains. We instantiate Blue-Pencil to three different domains: code, markdown, and spreadsheet transformations. We evaluated Blue-Pencil on 37 code (SQL and C#) editing sessions and found it suggested repetitive edits to the user within 199 ms on average. Furthermore, all Blue-Pencil's errors were transient – it returned the desired transformation on all of our benchmarks when it was not run on partially-completed edits. Our experiments suggest that Blue-Pencil can be used to suggest repetitive edits on the fly. Finally, we performed a field study to collect qualitative feedback on Blue-Pencil from professional programmers. The field study not only confirmed the viability of Blue-Pencil but also guided us in making several improvements to make the technology more robust.

As future work, we plan to conduct user studies to provide additional evidence to Blue-Pencil's usability, particularly over existing modal PBE systems. Additionally, we plan to design extensions based on it where the user can apply the suggestions over the entire code base, save them, share with other collaborators, and provide other inputs to the system as negative examples. We plan to improve the underlying PBE systems to increase the expressivity of Blue-Pencil and instantiate it to other domains, such as text (e.g., Word) and presentation (e.g., PowerPoint) documents. We hope that, with our future work, this technology can be used to produce the first applications that generate repetitive edit suggestions on the fly just by learning from users as they edit a document in a mode-less environment.

ACKNOWLEDGMENTS

The authors would like to thank all members of the Prose team at Microsoft, Mark Wilson-Thomas and his colleagues from Visual Studio IntelliCode, Microsoft developers who participated in evaluation studies, and Alex Polozov for their valuable feedback.

REFERENCES

J. Andersen and J. L. Lawall. 2008. Generic Patch Inference. In *Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering (ASE '08)*. IEEE Computer Society, Washington, DC, USA, 337–346. https://doi.org/10.1109/ASE.2008.44

Jesper Andersen, Anh Cuong Nguyen, David Lo, Julia L. Lawall, and Siau-Cheng Khoo. 2012. Semantic Patch Inference. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering (ASE 2012)*. ACM, New York, NY, USA, 382–385. https://doi.org/10.1145/2351676.2351753

CommonMark. 2019. CommonMark MarkDown Specification. (2019). At https://commonmark.org/.

Allen Cypher (Ed.). 1993. *Watch What I Do: Programming by Demonstration.* MIT Press.

Eclipse Foundation. 2019. Eclipse. (2019). At https://www.eclipse.org/.

Darren Edge, Sumit Gulwani, Natasa Milic-Frayling, Mohammad Raza, Reza Adhitya Saputra, Chao Wang, and Koji Yatani. 2015. Mixed-Initiative Approaches to Global Editing in Slideware. In *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems (CHI '15)*. ACM, New York, NY, USA, 3503–3512. https://doi.org/10.1145/2702123.2702551

John K. Feser, Swarat Chaudhuri, and Isil Dillig. 2015. Synthesizing Data Structure Transformations from Input-output Examples. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '15)*. ACM, New York, NY, USA, 229–239. https://doi.org/10.1145/2737924.2737977

J. Nathan Foster, Michael B. Greenwald, Jonathan T. Moore, Benjamin C. Pierce, and Alan Schmitt. 2007. Combinators for Bidirectional Tree Transformations: A Linguistic Approach to the View-update Problem. *ACM Trans. Program. Lang. Syst.* 29, 3, Article 17 (May 2007). https://doi.org/10.1145/1232420.1232424

Stephen R. Foster, William G. Griswold, and Sorin Lerner. 2012. WitchDoctor: IDE Support for Real-time Auto-completion of Refactorings. In *Proceedings of the 34th International Conference on Software Engineering (ICSE '12)*. IEEE Press, Piscataway, NJ, USA, 222–232. http://dl.acm.org/citation.cfm?id=2337223.2337250

GNU Emacs. 2019. Repeat Commands. https://www.gnu.org/software/emacs/manual/html_node/efaq/Repeating-commands.html. Accessed: 04/03/2019.

Sumit Gulwani. 2011. 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, 317–330.

JetBrains. 2019a. IntelliJ. (2019). At https://www.jetbrains.com/idea/.

JetBrains. 2019b. ReSharper. (2019). At https://www.jetbrains.com/resharper/.

Lingxiao Jiang, Ghassan Misherghi, Zhendong Su, and Stephane Glondu. 2007. DECKARD: scalable and accurate tree-based detection of code clones. 96–105. https://doi.org/10.1109/ICSE.2007.30

Miryung Kim and David Notkin. 2009. Discovering and Representing Systematic Code Changes. In *Proceedings of the 31st International Conference on Software Engineering (ICSE '09)*. IEEE Computer Society, Washington, DC, USA, 309–319. https://doi.org/10.1109/ICSE.2009.5070531

Miryung Kim, Thomas Zimmermann, and Nachiappan Nagappan. 2012. 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)*. ACM, New York, NY, USA, Article 50, 11 pages. https://doi.org/10.1145/2393596.2393655

Tessa Lau. 2001. Programming by Demonstration: a Machine Learning Approach.

Tessa Lau. 2008. Why PBD systems fail: Lessons learned for usable AI.

Tessa Lau, Steven A. Wolfman, Pedro Domingos, and Daniel S. Weld. 2001. Your Wish is My Command. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, Chapter Learning Repetitive Text-editing Procedures with SMARTedit, 209–226. http://dl.acm.org/citation.cfm?id=369505.369519

Vu Le and Sumit Gulwani. 2014. FlashExtract: A Framework for Data Extraction by Examples. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '14)*. New York, NY, USA, 542–553.

Alan Leung, John Sarracino, and Sorin Lerner. 2015. Interactive Parser Synthesis by Example. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '15)*. ACM, New York, NY, USA, 565–574. https://doi.org/10.1145/2737924.2738002

H. Lieberman. 2001. *Your Wish Is My Command: Programming by Example.* Morgan Kaufmann.

Michael Martin, Benjamin Livshits, and Monica S. Lam. 2005. Finding Application Errors and Security Flaws Using PQL: A Program Query Language. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA '05)*. ACM, New York, NY, USA, 365–383. https://doi.org/10.1145/1094811.1094840

Na Meng, Miryung Kim, and Kathryn S. McKinley. 2011. Systematic Editing: Generating Program Transformations from an Example. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '11)*. ACM, New York, NY, USA, 329–342.

Na Meng, Miryung Kim, and Kathryn S. McKinley. 2013. LASE: Locating and Applying Systematic Edits by Learning from Examples. In *Proceedings of the 35th International Conference on Software Engineering (ICSE '13)*. IEEE Press, Piscataway, NJ, USA, 502–511.

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

Microsoft. 2019a. Create or run a macro. https://support.office.com/en-us/article/Create-or-run-a-macro-C6B99036-905C-49A6-818A-DFB98B7C3C9C. Accessed: 04/03/2019.

Microsoft. 2019b. Microsoft PowerPoint. (2019). At https://products.office.com/en-us/powerpoint.

Microsoft. 2019c. Microsoft Word. (2019). At https://products.office.com/en-us/word.

Microsoft. 2019d. Refactoring source code in Visual Studio Code. (2019). At https://code.visualstudio.com/docs/editor/refactoring#_code-actions-quick-fixes-and-refactorings.

Microsoft. 2019e. Visual Studio. (2019). At https://www.visualstudio.com.

Robert C. Miller and Brad A. Myers. 2002. LAPIS: Smart editing with text structure. In *CHI '02*. ACM, 496–497.

Emerson Murphy-Hill and Andrew P. Black. 2007. High Velocity Refactorings in Eclipse. In *Proceedings of the 2007 OOPSLA Workshop on Eclipse Technology eXchange (eclipse '07)*. ACM, New York, NY, USA, 1–5. https://doi.org/10.1145/1328279.1328280

E. Murphy-Hill, C. Parnin, and A. P. Black. 2009. How we refactor, and how we know it. In *2009 IEEE 31st International Conference on Software Engineering*. 287–297. https://doi.org/10.1109/ICSE.2009.5070529

Stas Negara, Mihai Codoban, Danny Dig, and Ralph E. Johnson. 2014. Mining Fine-grained Code Changes to Detect Unknown Change Patterns. In *Proceedings of the 36th International Conference on Software Engineering (ICSE 2014)*. ACM, New York, NY, USA, 803–813. https://doi.org/10.1145/2568225.2568317

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

Hoan Anh Nguyen, Tung Thanh Nguyen, Gary Wilson, Jr., Anh Tuan Nguyen, Miryung Kim, and Tien N. Nguyen. 2010. A Graph-based Approach to API Usage Adaptation. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA '10)*. ACM, New York, NY, USA, 302–321. https://doi.org/10.1145/1869459.1869486

William F. Opdyke. 1992. *Refactoring Object-oriented Frameworks*. Ph.D. Dissertation. Champaign, IL, USA. UMI Order No. GAX93-05645.

Saswat Padhi, Prateek Jain, Daniel Perelman, Oleksandr Polozov, Sumit Gulwani, and Todd Millstein. 2018. FlashProfile: A Framework for Synthesizing Data Profiles. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 150 (Oct. 2018), 28 pages. https://doi.org/10.1145/3276520

J. Park, M. Kim, B. Ray, and D. Bae. 2012. An empirical study of supplementary bug fixes. In *2012 9th IEEE Working Conference on Mining Software Repositories (MSR)*. 40–49.

Oleksandr Polozov and Sumit Gulwani. 2015. FlashMeta: A Framework for Inductive Program Synthesis. In *Proceedings of the ACM International Conference on Object-oriented Programming Systems, Languages, and Applications (OOPSLA '15)*. ACM, New York, NY, USA, 542–553.

Veselin Raychev, Martin Vechev, and Eran Yahav. 2014. Code Completion with Statistical Language Models. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '14)*. ACM, New York, NY, USA, 419–428. https://doi.org/10.1145/2594291.2594321

Refsnes Data. 2019. XPath Tutorial. (2019). At https://www.w3schools.com/xml/xpath_intro.asp.

Reudismam Rolim, Gustavo Soares, Loris D'Antoni, Oleksandr Polozov, Sumit Gulwani, Rohit Gheyi, Ryo Suzuki, and Björn Hartmann. 2017. Learning Syntactic Program Transformations from Examples. In *Proceedings of the 39th International Conference on Software Engineering (ICSE '17)*. IEEE Press, Piscataway, NJ, USA, 404–415. https://doi.org/10.1109/ICSE.2017.44

Rishabh Singh and Sumit Gulwani. 2012. Synthesizing Number Transformations from Input-output Examples. In *Proceedings of the 24th International Conference on Computer Aided Verification (CAV'12)*. Springer-Verlag, Berlin, Heidelberg, 634–651. https://doi.org/10.1007/978-3-642-31424-7_44

Stack Exchange. 2019. How to programmatically edit all hyperlinks in a Word document? https://stackoverflow.com/q/3355266. Accessed: 04/03/2019.

Friedrich Steimann and Jens von Pilgrim. 2012. Refactorings Without Names. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering (ASE 2012)*. ACM, New York, NY, USA, 290–293. https://doi.org/10.1145/2351676.2351726

Vim. 2019. Macros. https://vim.fandom.com/wiki/Macros. Accessed: 04/03/2019.

Navid Yaghmazadeh, Xinyu Wang, and Isil Dillig. 2018. Automated Migration of Hierarchical Data to Relational Tables Using Programming-by-example. *Proc. VLDB Endow.* 11, 5 (Jan. 2018), 580–593. https://doi.org/10.1145/3187009.3177735

Pengcheng Yin, Graham Neubig, Miltiadis Allamanis, Marc Brockschmidt, and Alexander L. Gaunt. 2019. Learning to Represent Edits. In *International Conference on Learning Representations*. https://openreview.net/forum?id=BJl6AjC5F7