Overwatch: Learning Patterns in Code Edit Sequences

YUHAO ZHANG[*][†], University of Wisconsin-Madison, USA
YASHARTH BAJPAI[†], Microsoft, India
PRIYANSHU GUPTA[†], Microsoft, India
AMEYA KETKAR[*][†], Uber, USA
MILTIADIS ALLAMANIS, Microsoft Research, UK
TITUS BARIK, Microsoft, USA
SUMIT GULWANI, Microsoft, USA
ARJUN RADHAKRISHNA, Microsoft, USA
MOHAMMAD RAZA, Microsoft, USA
GUSTAVO SOARES, Microsoft, USA
ASHISH TIWARI, Microsoft, USA

Integrated Development Environments (IDEs) provide tool support to automate many source code editing tasks. Traditionally, IDEs use only the spatial context, i.e., the location where the developer is editing, to generate candidate edit recommendations. However, spatial context alone is often not sufficient to confidently predict the developer's next edit, and thus IDEs generate many suggestions at a location. Therefore, IDEs generally do not actively offer suggestions and instead, the developer is usually required to click on a specific icon or menu and then select from a large list of potential suggestions. As a consequence, developers often miss the opportunity to use the tool support because they are not aware it exists or forget to use it.

To better understand common patterns in developer behavior and produce better edit recommendations, we can additionally use the *temporal context*, i.e., the edits that a developer was recently performing. To enable edit recommendations based on temporal context, we present Overwatch, a novel technique for learning edit sequence patterns from traces of developers' edits performed in an IDE. Our experiments show that Overwatch has 78% precision and that Overwatch not only completed edits when developers missed the opportunity to use the IDE tool support but also predicted new edits that have no tool support in the IDE.

CCS Concepts: • **Software and its engineering** → **Software maintenance tools**; *Reusability*; • **Computing methodologies** → *Symbolic and algebraic algorithms*; *Unsupervised learning*.

Additional Key Words and Phrases: Program Generation, Artificial Intelligence, Program Synthesis

ACM Reference Format:
Yuhao Zhang, Yasharth Bajpai, Priyanshu Gupta, Ameya Ketkar, Miltiadis Allamanis, Titus Barik, Sumit Gulwani, Arjun Radhakrishna, Mohammad Raza, Gustavo Soares, and Ashish Tiwari. 2022. Overwatch:

[*] This work was done when these authors were employed at Microsoft
[†] Equal contribution

Authors' addresses: Yuhao Zhang, yuhaoz@cs.wisc.edu, University of Wisconsin-Madison, USA; Yasharth Bajpai, ybajpai@microsoft.com, Microsoft, India; Priyanshu Gupta, priyansgupta@microsoft.com, Microsoft, India; Ameya Ketkar, Uber, USA, ketkara@uber.com; Miltiadis Allamanis, Microsoft Research, UK, miltos@allamanis.com; Titus Barik, Microsoft, USA, tbarik@acm.org; Sumit Gulwani, Microsoft, USA, sumitg@microsoft.com; Arjun Radhakrishna, Microsoft, USA, arradha@microsoft.com; Mohammad Raza, Microsoft, USA, moraza@microsoft.com; Gustavo Soares, Microsoft, USA, gsoares@microsoft.com; Ashish Tiwari, Microsoft, USA, astiwar@microsoft.com.

2475-1421/2022/10-ART139
https://doi.org/10.1145/3563302

Learning Patterns in Code Edit Sequences. *Proc. ACM Program. Lang.* 6, OOPSLA2, Article 139 (October 2022), 29 pages. https://doi.org/10.1145/3563302

1 INTRODUCTION

Integrated Development Environments (IDEs) offer developers an overwhelming deluge of tools to support source code editing tasks, including writing new code, performing refactorings, and applying code fixes. Popular IDEs such as Microsoft Visual Studio [Microsoft 2021] and JetBrains ReSharper [JetBrains 2021], for example, provide over 100 C# refactorings, code fixes, and snippet tools. Traditionally, these tools use the location where the developer is editing code and the surrounding code as *spatial context* to generate candidate edits to recommend.

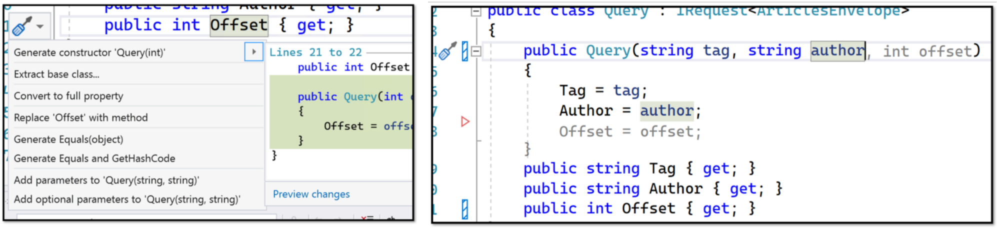
However, the spatial context alone is often not sufficient for IDEs to confidently predict the developer's next edit. At a specific location, there may be multiple candidate tools available for different editing tasks. For instance, Figure 1a shows all tools available when the developer clicks on the screwdriver next to a property declaration. There are 8 edits that the IDE can automate at that location. Unsurprisingly, developers have difficulty discovering these tools and applying them at the appropriate time and place [Ge et al. 2012; Murphy-Hill et al. 2009].

To improve code edit recommendations, in addition to the spatial context, we can also use the temporal context that the code edits which the developer was performing at a particular point in time. For instance, suppose the developer has just added the `Offset` property in Figure 1a. Next, the developer is more likely to add the corresponding parameter to the constructor and use it to initialize the property (7th option in the menu) than replace the nearly introduced property with a method (4th option). If the developer moves the cursor to the constructor, then it is very likely that they are about to insert the parameter. Recently, Visual Studio announced that they used this idea of temporal context to implement an analyzer to detect this *edit sequence* and offer the suggestion as "gray text" (Figure 1b) to add the parameter to the constructor as soon as the developer moves the cursor to the constructor after adding a new property. By using spatial and temporal contexts to generate suggestions at the right time and location, the IDE can afford to *preemptively* show these edit suggestions, avoiding *discoverability* (developers are unaware of the existing tool) and *late-awareness* problems (developers get further in their workflow before remembering an appropriate tool exists).

However, implementing tools that use temporal context is non-trivial. Tool builders have to reason not only about the location where an edit should be suggested and how to automate the edit but also how previous edits relate to the edit under consideration. Consider the example above, developers can perform the "Insert Property", "Insert Parameter", "Insert Assignment" edit sequence in any order but Visual Studio only handles the order shown in Figure 1. Given the complexity of manually implementing these *edit sequence patterns*, only few of them are available today in IDEs.

Instead of manually implementing patterns to recommend code edits, researchers have proposed several approaches to learn *edit patterns* from edits in source code repositories [Bader et al. 2019; de Sousa et al. 2021; Kim et al. 2013; Rolim et al. 2017; Yin et al. 2019]. These patterns represent the location where an edit should be applied and how to perform the desired edit. However, very few approaches use previous edits as temporal context. Blue-Pencil [Miltner et al. 2019] use previous edits to suggest similar repetitive edits. C^{3}PO [Brody et al. 2020] learns a model to complete an edit given other edits, but can only predict edits that do not generate new content. Additionally, C^{3}PO is trained on data from source code repositories, which do not capture the temporal context because the data do not reflect the order of edits made by developers in an IDE.

In this paper, we propose Overwatch, a technique for learning *Edit Sequence Patterns* from traces logged during editing sessions in the IDE. As input, Overwatch takes a set of source file versions. Each version represents the state of the file while a developer is editing it. Given this



```
    public string Author { get; }
    public int Offset { get; }
```

Generate constructor 'Query(int)'
Extract base class...
Convert to full property
Replace 'Offset' with method
Generate Equals(object)
Generate Equals and GetHashCode
Add parameters to 'Query(string, string)'
Add optional parameters to 'Query(string, string)'
Lines 21 to 22
Preview changes

```
    public int Offset

    public Query(int o
    {
        Offset = offs
    }
}
```

```
public class Query : IRequest<ArticlesEnvelope>
{
    public Query(string tag, string author, int offset)
    {
        Tag = tag;
        Author = author;
        Offset = offset;
    }
    public string Tag { get; }
    public string Author { get; }
    public int Offset { get; }
```

(a) Edit suggestions based on spatial context (b) Edit suggestion using spatial and temporal contexts

Fig. 1. Edits suggested by Visual Studio when the developer adds a property to a class

input, Overwatch's problem is to find recurrent edit sequences and generalize them into Edit Sequence Patterns (ESPs). In a nutshell, Overwatch performs three major steps: (1) generating edit sequence sketches and their corresponding specifications, (2) synthesizing edit sequence patterns, and (3) selecting and ranking the edit sequence patterns. Given a new development trace (i.e., edit history), Overwatch can then use the learned edit sequence patterns to predict the next edit.

To evaluate Overwatch, we collected 335, 687 source file versions, which were logged from 12 professional software developers from a large company across several months. In our experiments, Overwatch achieved 78.38% precision in the test set, showing a degree of domain-invariance, when compared to its performance on the validation set collected 6 months earlier. Additionally, we performed a qualitative analysis on the ESPs learned with Overwatch. Our findings show that ESPs can be used not only to complete edits when developers typically miss the opportunity to use the IDE tool support but also to predict new edits that have no tool support at all in the IDE. Finally, we show that Overwatch outperforms the closest approaches, C^{3}PO and Blue-Pencil, on the task of predicting the next edit in the edit sequences from our dataset.

In short, the paper makes the following contributions:

(1) We formalize the problem of learning Edit Sequence Patterns (ESPs) (Section 3);
(2) We propose Overwatch, a technique for learning edit sequences patterns from traces collected during editing sessions in the IDE (Sections 4-6);
(3) We show that the ESPs learned by Overwatch can be used to predict edits with 78.38% precision (Section 7.2);
(4) Our qualitative analysis shows that ESPs can be used not only to complete edits when developers missed the opportunity to use the IDE tool support but also predict new edits that have no tool support at all in the IDE (Section 7.3);
(5) Our experiments shows that Overwatch outperforms C^{3}PO and Blue-Pencil. While C^{3}PO does not support most of the edit sequences in our dataset, Blue-Pencil fails to synthesize transformations at the right level of abstraction in an offline setting (Section 7.4).

2 OVERVIEW

We begin with an overview of Overwatch's technique to learn ESPs and how we can use these patterns to predict code edits. To illustrate the process, we show how Overwatch learns an ESP that predicts the code edit recommended by Visual Studio in Figure 1b. As we mentioned, Visual Studio developers had to manually implement this feature, which is time-consuming and hard to scale. In Section 7.3 we present a list of other patterns that were automatically learned by Overwatch.

Consider the source file traces shown in Figures 2 and 3 depicting the sequences of versions produced when developers were performing similar edits in an IDE. At a high level, the developers are performing the same ESP: (a) adding a new property to the class, (b) adding a new parameter to

```
  class Node {

    Node() {
    }
  }
```

(a) v_0

```
  class Node {
+   public str Id { }
    Node() {
    }
  }
```

(b) v_1

```
  class Node {
-   public str Id { }
+   public str Id {get;}
    Node() {
    }
  }
```

(c) v_2

```
  class Node {
-   public str Id {get;}
+   public str Id {get;set;}
    Node() {
    }
  }
```

(d) v_3

```
  class Node {
    public str Id {get;set;}
-   Node() {
+   Node(str id) {

    }
  }
```

(e) v_4

```
  class Node {
    public str Id {get;set;}
    Node(str id) {
+     Id = id;
    }
  }
```

(f) v_5

Fig. 2. Development Session: Syntactically correct versions while adding and initializing a property.

```
class Graph {
  public int Id {get;set;}
  Graph(int id) {
    Id = id;
  }
}
```

(a) v_6

```
  class Graph {
    public int Id {get;set;}
+   public int Id {get;set;}
    Graph(int id) {
      Id = id;
    }
  }
```

(b) v_7

```
  class Graph {
    public int Id {get;set;}
-   public int Id {get;set;}
+   public int Size {get;set;}
    Graph(int id) {
      Id = id;
    }
  }
```

(c) v_8

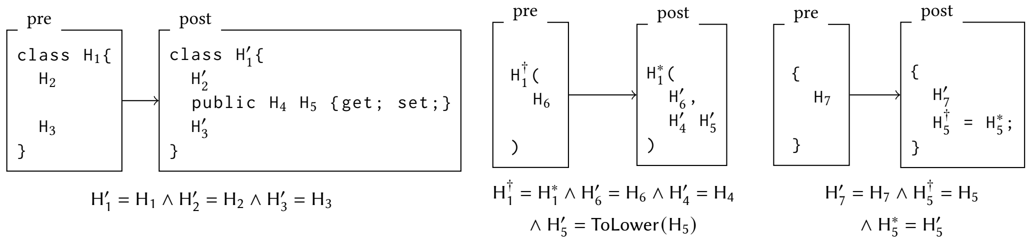
```
  class Graph {
    public int Id {get;set;}
    public int Size {get;set;}
-   Graph(int id) {
+   Graph(int id, int size) {
        Id = id;
    }
  }
```

(d) v_9

```
  class Graph {
    public int Id {get;set;}
    public int Size {get;set;}
    Graph(int id, int size) {
      Id = id;
+     Size = size;
    }
  }
```

(e) v_{10}

Fig. 3. Development Session: Syntactically correct versions while copying, updating, and initializing a property.



$H'_1 = H_1 \wedge H'_2 = H_2 \wedge H'_3 = H_3$

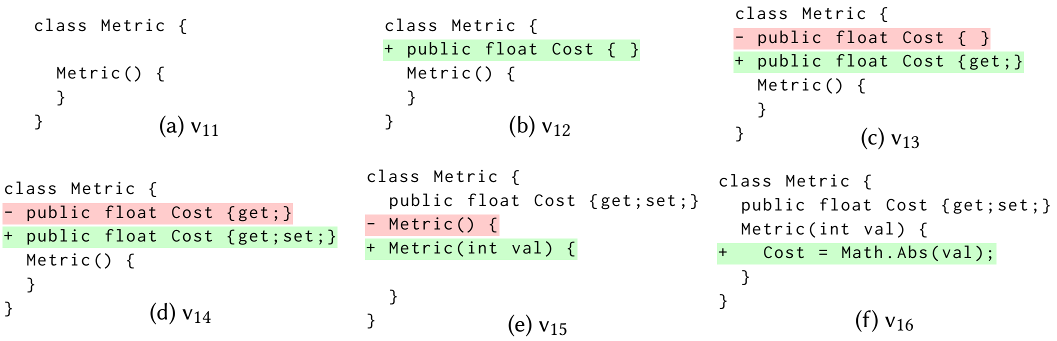
$H^\dagger_1 = H^*_1 \wedge H'_6 = H_6 \wedge H'_4 = H_4 \wedge H'_5 = \text{ToLower}(H_5)$

$H'_7 = H_7 \wedge H^\dagger_5 = H_5 \wedge H^*_5 = H'_5$

(a) InsertProperty (b) InsertConstructorParam (c) InsertAssignment

Fig. 4. Example of an Edit Sequence Pattern learned by OVERWATCH for the workflow InsertProperty · InsertConstructorParameter · InsertAssignment The variable component of the pattern (holes) are represented by H. Below each *pre* and *post* representaion of the template, we present the *Hole Predicates* specifying the relationship between holes across the edit pattern sequence.

the constructor with the same name as of the property (but lowercase) and same type, (c) adding a statement assigning the parameter to the property. However, the developers take different paths in the two cases—in Figure 2, the developer directly types in the new code while in Figure 3, the developer copies an existing property and changes the name. Figure 4 illustrates how OVERWATCH represents this pattern. Each individual transition represents the *pre* and *post* template of an edit template. We see that the *insert property* pattern in Figure 4a has templates with holes in it. Holes H_2 and H_3, respectively, represent the surrounding class members and methods preceding and



```
class Metric {

  Metric() {
  }
}
```

(a) v_{11}

```
class Metric {
+ public float Cost { }
  Metric() {
  }
}
```

(b) v_{12}

```
class Metric {
- public float Cost { }
+ public float Cost {get;}
  Metric() {
  }
}
```

(c) v_{13}

```
class Metric {
- public float Cost {get;}
+ public float Cost {get;set;}
  Metric() {
  }
}
```

(d) v_{14}

```
class Metric {
  public float Cost {get;set;}
- Metric() {
+ Metric(int val) {

  }
}
```

(e) v_{15}

```
class Metric {
  public float Cost {get;set;}
  Metric(int val) {
+   Cost = Math.Abs(val);
  }
}
```

(f) v_{16}

Fig. 5. Another sequence of versions that is different from the edit sequence pattern learned in Fig 4.

following the location of the edit. The *type* and *name* of the added property in the *post* template correspond to holes H_4 and H_5, respectively. Based on the newly added property, holes H_4 and H_5 can be replaced with the appropriate type and name to match the edit.

We use hole predicates to define relationships between holes in the pre- and post-templates. The predicates $\mathsf{H}'_i = \mathsf{H}_i$ for $i \in \{1, 2, 3\}$ represent that the class name and the class body does not change apart from the newly added property. Similarly, in Figure 4b, the predicate $\mathsf{H}'_6 = \mathsf{H}_6$ represents that the constructor parameters do not change except the newly added parameter. The predicate $\mathsf{H}'_5 = \mathsf{ToLower}(\mathsf{H}_5)$ says that the name of the parameter is the lower case version of the property name (e.g., if the property name is `Id`, the parameter name will be `id`). Note that this predicate relates the holes in two different edit templates, i.e., H_5 is in the InsertProperty template while H'_5 is in InsertConstructorParam. Hence, while learning an ESP, we need to consider the sequence of edits as a whole, instead of separately learning single edit patterns and putting them together.

2.1 Using Edit Sequence Patterns to Predict Edits

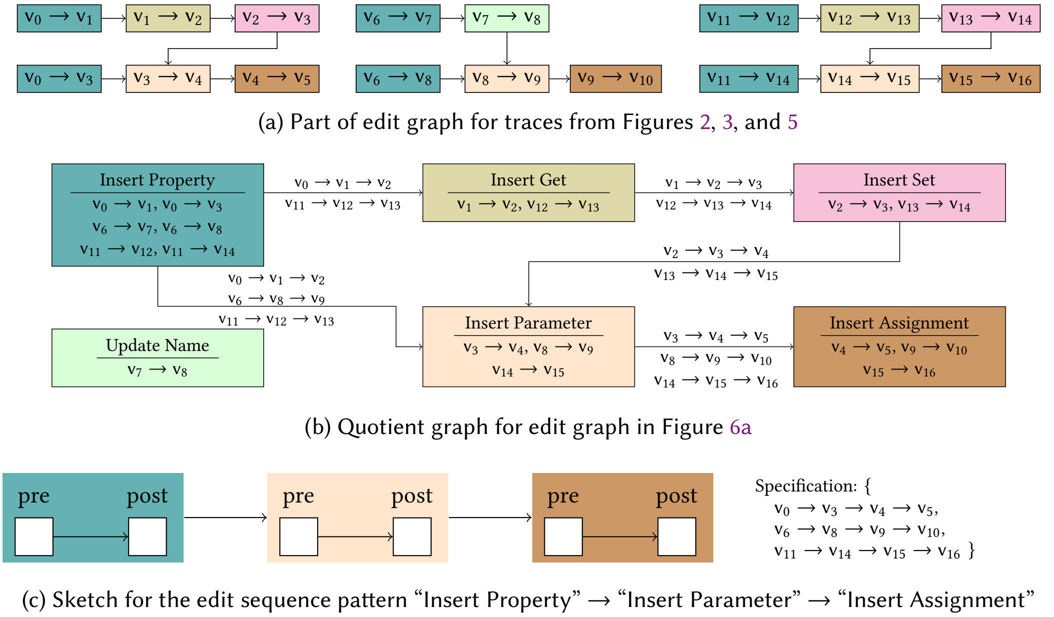
We can use the above pattern to predict the next edits that the developer will perform. For instance, consider the scenario shown in Figure 2. Suppose the developer has just performed the changes $v_0 \longrightarrow v_3$. We can match this edit to InsertProperty to get the values of the holes H_4 and H_5, i.e., `str` and `Id`, respectively. Now, using the predicates $\mathsf{H}'_4 = \mathsf{H}_4$ and $\mathsf{H}'_5 = \mathsf{ToLower}(\mathsf{H}_5)$, we can instantiate InsertConstructorParam to obtain the next edit. In an IDE, we can use this instantiation to suggest adding `str id` as soon as the developer moves the cursor to the constructor's parameter list using an interface similar to the one shown in Figure 1b. Note that in Figure 1b, we can predict two subsequent changes (adding the constructor parameter and adding an assignment) at once using edit patterns InsertConstructorParam and InsertAssignment in sequence.

Note that the predictions made using the ESP is just that, a prediction. As shown in Figure 5, the developer may actually want to make a different sequence of changes, i.e., the name and type of the property and the initialization expression are different that the ones predicted by the ESP. In our IDE plugin implementation, the developer can press the Escape key to ignore the recommendation from the edit sequence template and make their own change.

2.2 Learning Edit Sequence Patterns

Given the traces in Figures 2, 3, and 5 as input, Overwatch aims to learn the ESP in Figure 4.

Building the Edit Graph. Overwatch first creates the edit graph in Figure 6a to where the nodes represent edits at different *levels of granularity* and the directed edges represent *temporal relation*, i.e., one edit sequentially follows the other. For example, the figure contains both the node $v_0 \longrightarrow v_3$, as well as the nodes $v_0 \longrightarrow v_1$, $v_1 \longrightarrow v_2$, and $v_2 \longrightarrow v_3$; these represent the same change



(a) Part of edit graph for traces from Figures 2, 3, and 5

(b) Quotient graph for edit graph in Figure 6a

(c) Sketch for the edit sequence pattern "Insert Property" → "Insert Parameter" → "Insert Assignment"

Fig. 6. OVERWATCH: From Edit Graphs to Edit Sequence Patterns. We omit edits $v_0 \to v_2$ and $v_{11} \to v_{13}$ that should be in the edit graphs for ease of presentation.

of adding the property `public str Id { get; set; }`, but at different levels of granularity. The edit $v_0 \to v_3$ represents adding the full property, while $v_0 \to v_1$, $v_1 \to v_2$, and $v_2 \to v_3$ represent adding the property with the empty accessor list, adding the `get;`, and adding the `set;`. The edges between $v_0 \to v_1$, $v_1 \to v_2$, and $v_2 \to v_3$ represent that each edit immediately follows the previous in the trace. Note that the graph does not contain nodes for all changes (for example, $v_0 \to v_5$). We describe how we select the edits that should be there in the graph in Section 4–intuitively, we ignore large and unrelated edits.

Creating Sketches for Edit Pattern Sequences. Next, OVERWATCH produces a quotient graph by grouping together similar edits in the edit graph. Two edits are grouped together, i.e., in the same partition, if they have the same edit type (Insert, Delete, or Update) and the same type of AST node that is being modified (e.g., PropertyDeclaration and Parameter). In Figure 6a, the nodes are colored by partition. For example, the green nodes all represent the insertion of a PropertyDeclaration.

Figure 6b shows the quotient graph produced by OVERWATCH. The *quotient graph* summarizes the edit graph at the level of partitions: the vertices of the quotient graph are the partitions. An edge between two partitions exists in the quotient graph *iff* there are at least 2 pairs of edits in the partitions that sequentially follow each other. For example, there is an edge between InsertProperty and InsertParameter as there are 3 pairs of edits where a parameter is added immediately after a property is added (see edge label in Figure 6b). However, there are no edges to and from UpdateName since no two occurrences of update name are followed by edits of the same partition.

The paths in the quotient graph represent recurrent edit sequences applied by developers. For each path in the quotient graph, the *support* is the set of all edit sequences that correspond to it. For each path up to a size n with sufficient support in the quotient graph, OVERWATCH creates a sketch along with a *specification* that is given by its support. The right part of Figure 6c shows the sketch of the ESP *insert property*, *insert parameter*, and *assign property*, and the specification given by $\{\mathrm{edSeq}_1, \mathrm{edSeq}_2, \mathrm{edSeq}_3\}$ which correspond to the traces from Figures 2, 3, and 5.

From Sketches to Edit Sequence Patterns. In the next step, Overwatch uses these concrete sequences to infer edit templates and hole predicates to complete the sketch. We use a procedure based on anti-unification to generalize the edit sequences into edit templates and corresponding hole predicates. In essence, anti-unification is a technique to generalize two ASTs into a template by replacing differing subtrees with holes. However, we anti-unify edit sequences instead of ASTs and further, generate predicates relating the holes in the individual templates (see Section 5.2).

Anti-unifying the edit sequences edSeq_1, edSeq_2, and edSeq_3 produces the ESP depicted in Figure 4, but without the predicates $\mathsf{H}'_4 = \mathsf{H}_4$, $\mathsf{H}'_5 = \mathsf{ToLower}(\mathsf{H}_5)$, and $\mathsf{H}^*_5 = \mathsf{ToLower}(\mathsf{H}_5)$. This pattern, while general, cannot be used to predict the next changes. The absence of these predicates means that we can predict neither the name and type of the inserted parameter, nor the right-hand side of the assignment. On the contrary, anti-unifying just edSeq_1 and edSeq_2 produces exactly the pattern in Figure 4, which can be used for predictions as shown in Section 2.1. Overwatch uses agglomerative hierarchical clustering over edit sequences to produce a hierarchy of increasingly general ESPs. Hence, we will produce both ESPs (with and without anti-unifying edSeq_3). We select and rank a subset of the generated ESPs based on their predictive power on the input traces.

3 EDIT SEQUENCE PATTERNS

The goal of this paper is to learn a *sequence* of edit patterns from a set of developer edit traces and to make editing suggestions by the learned patterns while a developer is working in an IDE. In contrast to related works [Bader et al. 2019; de Sousa et al. 2021; Yin et al. 2019] that learn only a single edit pattern, we aim to use the hole predicates among the sequence of edit patterns. In this section, we show a novel representation for the sequence of edit patterns learned by our approach.

Versions and Development Sessions. A *version* v is a syntactically correct source file that occurs while a developer is editing code. A *development session* or *trace* $\mathsf{Trace} = \mathsf{v}_0 \ldots \mathsf{v}_n$ is the sequence of all versions that appear during an editing session. Here, we identify each version with its *abstract syntax tree* (AST). Hence, unparsable intermediate versions of code do not appear in the trace.

Edits and Edit Sequences. The edit $\mathsf{ed} = \mathsf{v}_{\mathsf{pre}} \longrightarrow \mathsf{v}_{\mathsf{post}}$ changes version $\mathsf{v}_{\mathsf{pre}}$ to $\mathsf{v}_{\mathsf{post}}$. The function $\mathsf{Localize}$ on edits that produce the smallest difference between the two ASTs in the edit. Formally, $\mathsf{Localize}(\mathsf{v}_{\mathsf{pre}} \longrightarrow \mathsf{v}_{\mathsf{post}}) = \mathsf{v}^*_{\mathsf{pre}} \longrightarrow \mathsf{v}^*_{\mathsf{post}}$ if: (a) $\mathsf{v}^*_{\mathsf{pre}}$ and $\mathsf{v}^*_{\mathsf{post}}$ are subtrees of $\mathsf{v}_{\mathsf{pre}}$ and $\mathsf{v}_{\mathsf{post}}$, respectively; (b) replacing $\mathsf{v}^*_{\mathsf{pre}}$ by $\mathsf{v}^*_{\mathsf{post}}$ in $\mathsf{v}_{\mathsf{pre}}$ yields $\mathsf{v}_{\mathsf{post}}$; and (c) $\mathsf{v}^*_{\mathsf{pre}}$ is the smallest subtree of such kind.

Example 3.1. Consider the edit $\mathsf{v}_3 \longrightarrow \mathsf{v}_4$ in Figure 2, where the developer adds the parameter `str id` to the constructor of `Node`. The localized version of this edit $\mathsf{Localize}(\mathsf{v}_3 \longrightarrow \mathsf{v}_4)$ is given by $\mathsf{v}^*_3 \longrightarrow \mathsf{v}^*_4$ where: (a) v^*_3 corresponds to the subtree of v_3 that represents the empty parameter list `()`, and (b) v^*_4 corresponds to the subtree of v_4 that represents the parameter list `(str id)`. □

An edit in the trace $\mathsf{Trace} = \mathsf{v}_0 \ldots \mathsf{v}_n$ is given by $\mathsf{v}_i \longrightarrow \mathsf{v}_j \in \mathsf{Edits}(\mathsf{Trace})$ where $0 \leq i < j \leq n$. Given edits $\mathsf{ed} = \mathsf{v}_i \longrightarrow \mathsf{v}_j$ and $\mathsf{ed}' = \mathsf{v}_k \longrightarrow \mathsf{v}_\ell$ from Trace, we say that ed' *sequentially follows* ed if $i < j = k < l$. This is written as $\mathsf{ed} \longrightarrow_{\mathsf{seq}} \mathsf{ed}'$. An *edit sequence* $\mathsf{ed}_0 \ldots \mathsf{ed}_n$ is a sequence of contiguous edits, i.e., $\forall i.\mathsf{ed}_i \longrightarrow_{\mathsf{seq}} \mathsf{ed}_{i+1}$.

Templates and Edit Templates. An *AST template* (or *template* for short) t is an AST where some leaf nodes are *holes*, i.e., they do not represent a program fragment but are placeholders. A *substitution* σ is a function that maps each hole to a finite sequence of AST nodes. The AST obtained by replacing each hole H in t by the node sequence $\sigma(\mathsf{H})$ is written as $\sigma(\mathsf{t})$. We assume that holes are unique, i.e., that a single hole does not appear in more than one location in a template and that multiple templates cannot share holes.

Example 3.2. An example of a template t is `(H, str id)` where H is a hole. This template represents all parameter lists of length 1 or more where the last parameter is `str id`. Note that we are writing templates using the equivalent code for readability. The template t is represented as an AST and does not contain a node for the comma separator.

With the substitution $\sigma_0 = \{\mathsf{H} \mapsto \epsilon\}$ that maps H to the empty sequence of nodes, we have $\sigma_0(\mathsf{t})$ = `(str id)`. Note that the comma disappears when we substitute the hole with the empty list—this is an artifact of writing the template as code. For $\sigma_1 = \{\mathsf{H} \mapsto$ `int count`$\}$ and $\sigma_2 = \{\mathsf{H} \mapsto$ `int count, str attr`$\}$ we have $\sigma_1(\mathsf{t})$ = `(int count, str id)` and $\sigma_2(\mathsf{t})$ = `(int count, str attr, str id)`. Note that σ_0, σ_1, and σ_2 map H to sequences of AST nodes of length 0, 1, and 2, respectively. □

We represent common editing motifs using *edit templates*. Formally, an edit template et = $\mathsf{t}_{\mathsf{pre}} \rightarrow \mathsf{t}_{\mathsf{post}}$ is a pair of AST templates. We say that an edit $\mathsf{v}_{\mathsf{pre}} \rightarrow \mathsf{v}_{\mathsf{post}}$ *matches* an edit template $\mathsf{t}_{\mathsf{pre}} \rightarrow \mathsf{t}_{\mathsf{post}}$ if: (a) $\mathsf{Localize}(\mathsf{v}_{\mathsf{pre}} \rightarrow \mathsf{v}_{\mathsf{post}}) = \mathsf{v}^*_{\mathsf{pre}} \rightarrow \mathsf{v}^*_{\mathsf{post}}$, and (b) there exists a substitution σ such that $\mathsf{v}^*_{\mathsf{pre}} = \sigma(\mathsf{t}_{\mathsf{pre}})$ and $\mathsf{v}^*_{\mathsf{post}} = \sigma(\mathsf{t}_{\mathsf{post}})$.

Example 3.3. An example of an edit template is et = $(\mathsf{H}_1) \rightarrow (\mathsf{H}_2,$ `str id`$)$. Here, the first template matches all parameter lists while the second matches all parameter lists where the last parameter is `str id`. Hence, it would match the edit $\mathsf{v}_3 \rightarrow \mathsf{v}_4$ in Figure 2. However, note that this edit template does not relate the values of H_1 and H_2 in pre- and post-versions of the edit. Therefore, an edit like `(int id)` → `(str label, str id)` will match the edit template et. We solve this issue using hole predicates below.

Hole Predicates. We introduce the notion of *hole predicates* to (a) relate the values of holes across multiple templates, and (b) restrict the set of substitutions that can be applied to a template. Formally, a hole predicate is an expression of type Boolean over holes and is evaluated over a substitution σ.

- **Unary predicates.** The predicate $\mathsf{IsNotNull}(\mathsf{H})$ asserts that the hole H cannot be replaced by an empty sequence, i.e., the substitution σ must satisfy $\sigma(\mathsf{H}) \neq \epsilon$ if $\mathsf{IsNotNull}(\mathsf{H}) = \mathsf{True}$. Another unary predicate $\mathsf{IsKind}_{\mathsf{label}}(\mathsf{H})$ is parametrized by an AST node type label (e.g., AssignExpr or ClassDeclaration). We have that $\mathsf{IsKind}_{\mathsf{label}}(\mathsf{H}) = \mathsf{True}$ for a substitution σ only if $\sigma(\mathsf{H}) = \mathsf{node}$ and the label of node is label. Note that $\mathsf{IsKind}_{\mathsf{kind}}$ forbids the hole value from being an empty sequence and a sequence with multiple elements.
- **Binary predicates.** We also use a class of predicates over two holes, written as $\mathsf{H}_1 = \mathsf{F}(\mathsf{H}_2)$ where F is a function. The most common F is the identity function in terms of text value, in which case, we write the predicate as $\mathsf{H}_1 = \mathsf{H}_2$. Other two functions F we use are ToLower and ToUpper, which indicate that the text value of H_1 in the substitution is the same as that of H_2, but the case of the first character changed appropriately.

Example 3.4 (Hole predicates). Consider the template t = `(H, str id)` from Example 3.2. Here, imposing the predicate $\mathsf{IsNotNull}(\mathsf{H})$ ensures that any AST matched by t must have at least 2 parameters in the parameter list. Continuing from Example 3.3, we can augment the edit template $(\mathsf{H}_1) \rightarrow (\mathsf{H}_2,$ `str id`$)$ with the hole predicate $\mathsf{H}_1 = \mathsf{H}_2$ to ensure that we exactly capture the class of edits that insert a new parameter `str id` to an existing parameter list. □

Example 3.5. Hole predicates can be used to relate holes across multiple edits to exactly capture the common editing pattern illustrated in Figure 2.

- *Add a new property to a class.* This category of edits is captured by the edit template et_1 = $\{\mathsf{H}_1\ \mathsf{H}_2\} \rightarrow \{\mathsf{H}_3$ `public` $\mathsf{H}_4\ \mathsf{H}_5$ `{get;}` $\mathsf{H}_6\}$. Here, H_1 and H_2 represent the class members that appear before and after the newly inserted property, respectively. The type and name of the property are represented by H_4 and H_5, respectively. We can add the unary predicates $\mathsf{IsKind}_{\mathsf{Type}}(\mathsf{H}_4)$ and $\mathsf{IsKind}_{\mathsf{Ident}}(\mathsf{H}_5)$ to ensure that H_4 and H_5 are a Type node and an Identifier node, respectively. To ensure that the contents of the class do not change apart from the newly inserted property, we need the predicates $\mathsf{H}_3 = \mathsf{H}_1$ and $\mathsf{H}_6 = \mathsf{H}_2$.

- *Add a new parameter to the constructor.* This edit is captured by the edit template and predicates similar to the ones presented in Example 3.4. We have $\mathsf{et}_2 = (\mathsf{H}_7) \to (\mathsf{H}_8,\ \ \mathsf{H}_9\ \ \mathsf{H}_{10})$ with the predicate $\mathsf{H}_8 = \mathsf{H}_7$ to ensure that the parameter list is preserved apart from the new parameter. We have the additional predicates $\mathsf{H}_9 = \mathsf{H}_4$ and $\mathsf{H}_{10} = \mathsf{ToLower}(\mathsf{H}_5)$ to ensure that the type and name match that of the inserted property. Note that the relation between H_5 and H_{10} is not strict equality, but involves an additional transformation ToLower to H_5.
- *Assign the new parameter to the new property.* These edits add a new assignment statement to the end of the block and are captured by $\mathsf{et}_3 = \{\mathsf{H}_{11}\} \to \{\mathsf{H}_{12};\ \ \mathsf{H}_{13} = \mathsf{H}_{14};\}$ with the predicates $\mathsf{H}_{12} = \mathsf{H}_{11}$, $\mathsf{H}_{13} = \mathsf{H}_5$, and $\mathsf{H}_{14} = \mathsf{H}_{10}$. However, we omit some unary predicates if the absence does not hinder understanding for ease of presentation in this paper.

Together, the edit templates et_1, et_2, and et_3 along with the above predicates fully capture the common editing pattern of adding a new property to a class and initializing it in the constructor. □

Edit Sequence Patterns. The main object of study in this paper is an *Edit Sequence Pattern* (ESP). ESPs are used to capture sequences of common editing actions like in Example 3.5. Formally, an ESP is a pair $\langle \mathsf{TS}, \mathsf{Preds} \rangle$ where: (a) TS is a restricted regular expression over edit templates, and (b) Preds is a set of hole predicates. Here, the restricted regular expression TS is of the form $\mathsf{et}_1 \ldots \mathsf{et}_{n-1}\mathsf{et}_n^{[*]}$ where $[*]$ represents an optional Kleene star. That is, TS is a sequence of edit templates where the last template may have a Kleene star.

Example 3.6. The edit templates and hole predicates from Example 3.5 can be written as an ESP $\langle \mathsf{TS}, \mathsf{Preds} \rangle$. Here, $\mathsf{TS} = \mathsf{et}_1\mathsf{et}_2\mathsf{et}_3$ and $\mathsf{Preds} = \{\mathsf{H}_3 = \mathsf{H}_1, \mathsf{H}_6 = \mathsf{H}_2, \mathsf{H}_8 = \mathsf{H}_7, \mathsf{H}_9 = \mathsf{H}_4, \mathsf{H}_{10} = \mathsf{ToLower}(\mathsf{H}_5), \mathsf{H}_{12} = \mathsf{H}_{11}, \mathsf{H}_{13} = \mathsf{H}_5, \mathsf{H}_{14} = \mathsf{H}_{10}\} \cup \{\mathsf{IsKind}_{\mathsf{Type}}(\mathsf{H}_4), \mathsf{IsKind}_{\mathsf{Identifier}}(\mathsf{H}_5), \ldots\}$.

```
class Comms {
    // Edit 1
-   void Write(Stream s,
-   byte[] bs, bool flush) { }
+   void Write(Stream s,
+   byte[] bs) { }
}
void Main() {
    // Edit 2
-   Comms.Write(io, bytes, f);}
+   Comms.Write(io, bytes);
    // Edit 3
-   Comms.Write(io, result, f);
+   Comms.Write(io, result);
}
```

Fig. 7. Delete a Parameter and Delete Arguments

We say that a sequence of edits $\mathsf{ed}_1 \ldots \mathsf{ed}_n$ matches $\langle \mathsf{TS}, \mathsf{Preds} \rangle$, where $\mathsf{TS} = \mathsf{et}_1 \ldots \mathsf{et}_{n-1}\mathsf{et}_n$, if there exists a substitution σ such that: (a) each ed_i matches et_i for $1 \leq i \leq n$, (b) the hole valuations in σ satisfy all the predicates in Preds.

Extending this definition, we say that a sequence of edits $\mathsf{ed}_1 \ldots \mathsf{ed}_m$ (with $m \geq n$) matches $\langle \mathsf{TS}, \mathsf{Preds} \rangle$, where $\mathsf{TS} = \mathsf{et}_1 \ldots \mathsf{et}_{n-1}\mathsf{et}_n^*$, if each of the sequences $\mathsf{ed}_1 \ldots \mathsf{ed}_{n-1}\mathsf{ed}_k$ for $n \leq k \leq m$ matches $\langle \mathsf{et}_1 \ldots \mathsf{et}_{n-1}\mathsf{et}_n, \mathsf{Preds} \rangle$.

Example 3.7. Consider an ESP $\langle \mathsf{TS}, \mathsf{Preds} \rangle$, where $\mathsf{TS} = \mathsf{et}_1\mathsf{et}_2^*$ has a Kleene star, with $\mathsf{et}_1 = (\mathsf{H}_1, \mathsf{H}_2\ \mathsf{H}_3) \to (\mathsf{H}_4)$, $\mathsf{et}_2 = (\mathsf{H}_5, \mathsf{H}_6) \to (\mathsf{H}_7)$, and $\mathsf{Preds} = \{\mathsf{H}_1 = \mathsf{H}_4, \mathsf{H}_5 = \mathsf{H}_7, \mathsf{IsKind}_{\mathsf{Type}}(\mathsf{H}_2), \mathsf{IsKind}_{\mathsf{Ident}}(\mathsf{H}_3), \mathsf{IsKind}_{\mathsf{Arg}}(\mathsf{H}_6)\}$. This pattern represents an editing sequence where the developer deletes the last parameter in a declaration, and then, deletes the corresponding argument in multiple callsites.

Consider the three edits ed_1, ed_2, and ed_3 in Figure 7. We have that $\mathsf{ed}_1\mathsf{ed}_2\mathsf{ed}_3$ matches $\mathsf{et}_1\mathsf{et}_2^*$. To show this, we need to show that both $\mathsf{ed}_1\mathsf{ed}_2$ and $\mathsf{ed}_1\mathsf{ed}_3$ match the un-starred ESP $\langle \mathsf{et}_1\mathsf{et}_2, \mathsf{Preds} \rangle$. We can see that $\mathsf{ed}_1\mathsf{ed}_2$ matches $\mathsf{et}_1\mathsf{et}_2$ with the substitution $\sigma_1 = \{\mathsf{H}_1 \mapsto \texttt{Stream s, byte[] bs}, \mathsf{H}_2 \mapsto \texttt{bool}, \mathsf{H}_3 \mapsto \texttt{flush}, \mathsf{H}_4 \mapsto \texttt{Stream s, byte[] bs}, \mathsf{H}_5 \mapsto \texttt{io, bytes}, \mathsf{H}_6 \mapsto \texttt{f}, \mathsf{H}_7 \mapsto \texttt{io, bytes}\}$. Similarly, we can show that $\mathsf{ed}_1\mathsf{ed}_3$ matches $\mathsf{et}_1\mathsf{et}_2$ with the substitution σ_2, which is the same as σ_1 with `bytes` replaced by `result` for H_5 and H_7. □

Remark 3.8. In our implementation, we consider a slightly more general form of edit sequence patterns. There, we can have ESPs where any edit template (not just the last one) may be starred. These more general patterns can be formalized in a straightforward way, though we do not do so here for ease of presentation.

Using Edit Sequence Patterns. After a edit sequence matches a prefix of an ESP, we can use the next edit template in the ESP to predict the next change that the developer will make. We illustrate an usage of an ESP in the following example and we will further describe the details in Section 6.

Example 3.9 (Usage of an ESP). Consider the ESP $\langle \mathsf{et}_1\mathsf{et}_2\mathsf{et}_3, \mathsf{Preds}\rangle$ defined in Example 3.6, and the edit sequence $\mathsf{ed}_1 \rightarrow_{\mathsf{seq}} \mathsf{ed}_2$ from Figure 2, where $\mathsf{ed}_1 = \mathsf{v}_0 \rightarrow \mathsf{v}_3, \mathsf{ed}_2 = \mathsf{v}_3 \rightarrow \mathsf{v}_4$. We will consider the task of predicting the next edit give that the developer has just performed ed_1 and ed_2.

- First, we find a substitution σ such that ed_1 and ed_2 match et_1 and et_2, respectively using σ. Further, we require that σ satisfies each predicate in Preds that is over only the holes appearing in et_1 and et_2. Here, we have $\sigma = \{\mathsf{H}_2 \mapsto \mathtt{Node()\ \{\ \}}, \mathsf{H}_4 \mapsto \mathtt{str}, \mathsf{H}_5 \mapsto \mathtt{Id}, \mathsf{H}_6 \mapsto \mathtt{Node()\ \{\ \}}, \mathsf{H}_9 \mapsto \mathtt{str}, \mathsf{H}_{10} \mapsto \mathtt{id}, \} \cup \{\mathsf{H}_i \mapsto \epsilon \mid i \in \{1, 3, 7, 8\}\}$.
- Then, we find an AST node in v_4 such that the node matches $\mathsf{et}_{3,\mathsf{pre}}$ using a substitution σ'. We get $\sigma' = \{\mathsf{H}_{11} \mapsto \epsilon\}$ for the AST node that represents the empty body of the constructor. And we require that $\sigma \cup \sigma'$ satisfies all predicates in Preds that are over the domain of $\sigma \cup \sigma'$.
- Now, we use the predicates in Preds that contain the holes from $\mathsf{et}_{3,\mathsf{post}}$ to predict the values for those holes. Here, from the predicates $\mathsf{H}_{12} = \mathsf{H}_{11}$, $\mathsf{H}_{13} = \mathsf{H}_5$, and $\mathsf{H}_{14} = \mathsf{H}_{10}$, we can predict that $\mathsf{H}_{12} \mapsto \epsilon$, $\mathsf{H}_{13} \mapsto \mathtt{Id}$, and $\mathsf{H}_{14} = \mathtt{id}$.
- Filling in these values in $\mathsf{et}_{3,\mathsf{post}}$, we get the new constructor body `{Id = id;}`. The predicted version is obtained by replacing node in v_4 with this new constructor body. This exactly produces the version v_5 in Figure 2.

Remark 3.10. Intuitively, ESPs are a mechanism for predicting the next edit based on the *temporal context*, i.e., the sequence of atomic edits the developer has been performing. However, the ESPs themselves may operate over the non-atomic edits, i.e., they may match coarse-grained non-atomic edits in a session. Further, as we will see below, the ESPs are learned by generalizing patterns in non-atomic edits over multiple sessions from different developers.

Problem Statement and Solution Sketch. The input to the *ESP learning* problem is a set of traces. The expected output is a ranked set of ESPs $\langle \mathsf{TS}_1, \mathsf{Preds}_1\rangle \ldots \langle \mathsf{TS}_n, \mathsf{Preds}_n\rangle$. The aim is to produce ESPs that are helpful in predicting the next version in any trace. To this end, we measure the quality of the output using the standard notions of *precision* and *recall*, and a general F score (see Section 7.2 for more details).

Our solution strategy is in 3 parts:

- *Generating edit sequence sketches and specifications.* (Section 4, Lines 1-6 in Algorithm 1) The first step is to generate sets of concrete edit sequences (called the specification) that can potentially all match the same ESP, along with a sketch for that ESP. To generate these sketches and specifications, we (a) partition the set of all edits in Traces (Lines 1-3), (b) summarize the edit graph by the partitions to build a quotient graph (Line 4), and (c) generate sketches and specifications from paths of the quotient graph (Lines 5-6).
- *Synthesizing ESPs.* (Section 5, Lines 7-9 in Algorithm 1) Given these edit sequence sketches and specifications, we generate a hierarchy of ESPs iteratively where each pattern in the hierarchy is more general and matches more edit sequences in the specifications than the patterns lower in the hierarchy. The core algorithm here takes as input a set of edit sequences and produces a set of ESPs that can potentially matches the provided edit sequences.

Algorithm 1 Overview of Overwatch

```
Require: Set of traces Traces
Ensure: Ranked list of ESPs
 1: edits ⟵ $\bigcup\{\text{Edits(Trace)} \mid \text{Trace} \in \text{Traces}\}$
 2: EditGraph ⟵ BuildEditGraph(edits)
 3: Partitions ⟵ partition of edits based on Kind
 4: QuotientGraph ⟵ Quotient(EditGraph, Partitions)
 5: Paths ⟵ FrequentPaths(QuotientGraph)
 6: SketchesAndSpecs ⟵ {GenerateSketchAndSpec(path) | path ∈ Paths}
 7: Patterns ⟵ ∅
 8: for (sk, spec) ∈ SketchesAndSpecs do
 9:     Patterns ⟵ Patterns ∪ LearnPatterns(sk, spec)
10: return FilterAndSelect(Patterns)
```

- *Selecting and ranking ESPs.* (Section 6, Line 10 in Algorithm 1) Once we build a hierarchy of ESPs, we determine their predictive power by testing them on the input Traces. Based on their precision on the Traces, we select a subset of the patterns and rank them accordingly.

4 FROM TRACES TO EDIT PATTERN SKETCHES

In this section, we produce sketches and specifications from a set of traces. Formally, an *edit pattern sketch* sk is of the form $A_1 \ldots A_{n-1}A_n^{[*]}$ where each A_i is a placeholder for an edit template. A *specification* spec for a sketch sk is a set of edit sequences such that the length of each edit sequence in spec (a) is equal to n if A_n is un-starred in sk, and (b) is at least n if A_n is starred in sk.

Example 4.1. Given a set of input traces that include the traces from Figures 2 and 3, the technique in this section will produce a set of pairs of the form (sk, spec). One such pair might be sk = $A_1A_2A_3$ and spec = $\{\mathsf{ed}_1\mathsf{ed}_2\mathsf{ed}_3, \mathsf{ed}'_1\mathsf{ed}'_2\mathsf{ed}'_3, \ldots\}$ where $\mathsf{ed}_1 = \mathsf{v}_0 \longrightarrow \mathsf{v}_3$, $\mathsf{ed}_2 = \mathsf{v}_3 \longrightarrow \mathsf{v}_4$, $\mathsf{ed}_3 = \mathsf{v}_4 \longrightarrow \mathsf{v}_5$, $\mathsf{ed}'_1 = \mathsf{v}_6 \longrightarrow \mathsf{v}_8$, $\mathsf{ed}'_2 = \mathsf{v}_8 \longrightarrow \mathsf{v}_9$, and $\mathsf{ed}'_3 = \mathsf{v}_9 \longrightarrow \mathsf{v}_{10}$. Note that (a) ed_1 and ed'_1 add a new property, (b) ed_2 and ed'_2 add a new parameter to the constructor, and (c) ed_3 and ed'_3 assign the newly added parameter to the newly added property. This sketch and specification will then be used in Section 5 to generate a hierarchy of ESPs. □

We synthesize sketches of ESPs from traces in three steps, (a) build an edit graph that contains information about the granularity and sequencing of edits in the input traces, (b) produce a summary of edit sequences by quotienting the edit graph based on a partitioning of edits, and (c) produce sketches and specifications of ESPs by finding frequent paths in the summary quotient graph. We explain each of these steps below.

Generating the Edit Graph. The edit graph represents all edits in all input traces, as a graph. First, we collect the set of all edits at all granularities in the input traces, i.e., edits between all pairs (not necessarily consecutive) of versions. Since the number of edits grows quadratically in the length of the trace, in practice, we prune the edits as follows. First, we *debounce* the transient edits because these edits are likely noisy, i.e., we delete edits where the two versions were separated by less than 500ms of time [Miltner et al. 2019]. Second, we remove edits where the change is *larger than a given threshold.* Large edits are likely to incorporate changes that are completely unrelated to each other. For example, the edit of adding a new class and implementing all its methods is likely to contain many unrelated edits, and not be a part of any common editing workflow. Now, the individual edits from this pruned set form the vertices of the graph and there is an edge between ed_1 and ed_2 if and only if ed_2 sequentially follows ed_1, i.e., $\mathsf{ed}_1 \longrightarrow_{\mathsf{seq}} \mathsf{ed}_2$. Note that the edit graph contains edits at different levels of granularity. For example, in the edit graph for a trace with versions $\mathsf{v}_0\mathsf{v}_1\mathsf{v}_2$, both the coarse-grained edit $\mathsf{v}_0 \longrightarrow \mathsf{v}_2$, as well as the fine-grained edits $\mathsf{v}_0 \longrightarrow \mathsf{v}_1$ and $\mathsf{v}_1 \longrightarrow \mathsf{v}_2$.

Example 4.2. The edit graph of the trace shown in Figure 2 contains vertices of the form $v_{ij} = \mathsf{v}_i \longrightarrow \mathsf{v}_j$ for $0 \leq i < j \leq 5$. The edit graph is shown in Figure 6a. Note that the graph contains nodes for both fine-grained edits $\mathsf{v}_0 \longrightarrow \mathsf{v}_1$, $\mathsf{v}_1 \longrightarrow \mathsf{v}_2$, $\mathsf{v}_2 \longrightarrow \mathsf{v}_3$, and $\mathsf{v}_3 \longrightarrow \mathsf{v}_4$, as well as the coarse-grained edit $\mathsf{v}_0 \longrightarrow \mathsf{v}_4$. There is an edge between $v_{03} \longrightarrow v_{34}$ as $\mathsf{v}_3 \longrightarrow \mathsf{v}_4$ sequentially follows $\mathsf{v}_0 \longrightarrow \mathsf{v}_3$. On the other hand, there is no edge from v_{03} to v_{45}.

Summarizing the Edit Graph. Once the edit graph is built, the next task is to create an abstract version of the edit graph that groups together edits of *similar* kind. To define similar, we first define an embedding of edits. We categorize edits into 3 types: insert, delete, and update. The edit *insert child* $\mathsf{Insert}(\mathsf{parent}, \mathsf{child}, i)$ and the edit *delete child* $\mathsf{Delete}(\mathsf{parent}, \mathsf{child}, i)$ insert and delete AST node child of parent's children at position i, respectively, whereas an *update* $\mathsf{Update}(\mathsf{old}, \mathsf{new})$ replaces the AST node old with new. Insert and delete child operations are also updates (of the parent parent); however, we assume edits are written as insert or delete child when possible.

Given an edit ed, we define the *kind of the edit* $\mathsf{Kind}(\mathsf{ed})$ to be $(\mathsf{operation}, \mathsf{label})$ where: (a) operation is one of Delete, Insert, or Update; and (b) label is the type of node that is being deleted, inserted, or updated (e.g. MethodInvocation or Identifier). In an edit graph, we call the set of all vertices (edits) of the same kind a *partition*.

Example 4.3. In Figure 6a, the edit $\mathsf{v}_0 \longrightarrow \mathsf{v}_3$ is of type $(\mathsf{Insert}, \mathsf{Property})$ and the type of $\mathsf{v}_3 \longrightarrow \mathsf{v}_4$ is $(\mathsf{Insert}, \mathsf{Parameter})$. The partition for $(\mathsf{Insert}, \mathsf{Property})$ is given by $\{\mathsf{v}_0 \longrightarrow \mathsf{v}_1, \mathsf{v}_0 \longrightarrow \mathsf{v}_2, \mathsf{v}_0 \longrightarrow \mathsf{v}_3, \mathsf{v}_6 \longrightarrow \mathsf{v}_7, \mathsf{v}_6 \longrightarrow \mathsf{v}_8, \mathsf{v}_{11} \longrightarrow \mathsf{v}_{12}, \mathsf{v}_{11} \longrightarrow \mathsf{v}_{13}, \mathsf{v}_{11} \longrightarrow \mathsf{v}_{14}, \}$.

The *quotient graph* of the edit graph summarizes the sequencing information present in the edit graph at the level of partitions. We build the quotient graph by lifting the edit graph's sequencing information to the level of partitions.

- *Quotient graph vertices.* A vertex in the quotient graph is a partition, i.e., the set of edits from the edit graph with the same Kind. We use the term Partitions to denote the set of all vertices in the quotient graph.
- *Quotient graph edges.* Classically, an edge exists between two vertices $\mathsf{P} \longrightarrow \mathsf{P}'$ in the quotient graph when there exist $\mathsf{ed} \in \mathsf{P}, \mathsf{ed}' \in \mathsf{P}'$ with an edge $\mathsf{ed} \longrightarrow_{\mathsf{seq}} \mathsf{ed}'$ between them (see, for example, [Bloem et al. 2006]). Here, we strengthen the requirement by asking at least s different pairs of such ed and ed$'$. This ensures that the ESPs we generate are general, i.e., eliminating patterns corresponding to the editing mannerisms and habits particular to individual developers. In our experiments, we use $s = 2$.
- *Quotient graph edge labels.* We associate each edge $\mathsf{P} \longrightarrow \mathsf{P}'$ in the quotient graph with a label $\mathsf{Label}(\mathsf{P} \longrightarrow \mathsf{P}')$ that is a set of edit pairs. We define $\mathsf{Label}(\mathsf{P} \longrightarrow \mathsf{P}')$ to be $\{(\mathsf{ed}, \mathsf{ed}') \mid \mathsf{ed} \in \mathsf{P}, \mathsf{ed}' \in \mathsf{P}, \mathsf{ed} \longrightarrow_{\mathsf{seq}} \mathsf{ed}'\}$, i.e., it contains all pairs of contiguous edits.

Example 4.4. The quotient graph for the edit graph in Figure 6a is shown in Figure 6b. There are 3 different vertices (partitions) P_1, P_2, and P_3 in the quotient graph corresponding to the kinds $(\mathsf{Insert}, \mathsf{Property})$, $(\mathsf{Insert}, \mathsf{Parameter})$, and $(\mathsf{Insert}, \mathsf{Assignment})$, respectively. There is an edge $\mathsf{P}_1 \longrightarrow \mathsf{P}_2$ from P_1 to P_2 as there are $3 > s$ corresponding sequentially consecutive edit pairs: (a) $\mathsf{v}_0 \longrightarrow \mathsf{v}_3$ and $\mathsf{v}_3 \longrightarrow \mathsf{v}_4$, (b) $\mathsf{v}_6 \longrightarrow \mathsf{v}_8$ and $\mathsf{v}_8 \longrightarrow \mathsf{v}_9$, and (c) $\mathsf{v}_{11} \longrightarrow \mathsf{v}_{14}$ and $\mathsf{v}_{14} \longrightarrow \mathsf{v}_{15}$. Further, the label $\mathsf{Label}(\mathsf{P}_1 \longrightarrow \mathsf{P}_2)$ is given the same set of 3 edits.

Generating Sketches and Specifications. From the quotient graph, we generate ESP sketches and corresponding specifications paths using paths in the quotient graph. First, we define the support $\mathsf{Support}(\mathsf{P}_1 \ldots \mathsf{P}_n)$ as follows: (a) For the path with only 1 edge, we define $\mathsf{Support}(\mathsf{P}_1\mathsf{P}_2)$

to be the label $\mathsf{Label}(\mathsf{P}_1 \longrightarrow \mathsf{P}_2)$, and (b) Otherwise, we define $\mathsf{Support}(\mathsf{P}_1\mathsf{P}_2 \dots \mathsf{P}_n)$ recursively as $\{\mathsf{ed}_1\mathsf{ed}_2 \dots \mathsf{ed}_n \mid (\mathsf{ed}_1 \longrightarrow \mathsf{ed}_2) \in \mathsf{Support}(\mathsf{P}_1, \mathsf{P}_2), (\mathsf{ed}_2\mathsf{ed}_3 \dots \mathsf{ed}_n) \in \mathsf{Support}(\mathsf{P}_2 \dots \mathsf{P}_n)\}$.

Now, we define a *frequent path* in the quotient graph as any path $\mathsf{P}_1 \dots \mathsf{P}_n$ where $\mathsf{Support}(\mathsf{P}_1 \dots \mathsf{P}_n)$ has cardinality greater than a threshold $s = 2$. The set of frequent paths can be computed recursively by starting with single edges and adding edges to the end as long as the support is greater than the threshold.

From the set of frequent paths, we generate two different kinds of sketch specification pairs.

- For any *simple* path $\mathsf{P}_1 \dots \mathsf{P}_n$ (i.e., satisfying $i \neq j \implies \mathsf{P}_i \neq \mathsf{P}_j$), we define $\mathsf{sk} = A_1 \dots A_n$ and $\mathsf{spec} = \mathsf{Support}(\mathsf{P}_1 \dots \mathsf{P}_n)$.
- For a set of paths $\{\mathsf{P}_1 \dots \mathsf{P}_{n-1}\mathsf{P}_n, \mathsf{P}_1 \dots \mathsf{P}_{n-1}\mathsf{P}_n\mathsf{P}_n, \dots, \mathsf{P}_1 \dots \mathsf{P}_{n-1}\mathsf{P}_n^k\}$ where $\mathsf{P}_1 \dots \mathsf{P}_{n-1}$ is simple, we define $\mathsf{sk} = A_1 \dots A_n^*$ and $\mathsf{spec} = \bigcup_{1 \leq i \leq k} \mathsf{Support}(\mathsf{P}_1 \dots \mathsf{P}_{n-1}\mathsf{P}_n^i)$.

Example 4.5. One possible frequent paths in Figure 6b are given by $\mathsf{P}_1\mathsf{P}_2\mathsf{P}_3$ where the partitions are equivalent to insert property, insert parameter, and insert assignment as described in Example 4.4. From this path, we generate the sketch $\mathsf{sk} = A_1A_2A_3$ and the specification $\mathsf{spec} = \{(\mathsf{v}_0 \longrightarrow \mathsf{v}_3)(\mathsf{v}_3 \longrightarrow \mathsf{v}_4)(\mathsf{v}_4 \longrightarrow \mathsf{v}_5), (\mathsf{v}_6 \longrightarrow \mathsf{v}_8)(\mathsf{v}_8 \longrightarrow \mathsf{v}_9)(\mathsf{v}_9 \longrightarrow \mathsf{v}_{10}), (\mathsf{v}_{11} \longrightarrow \mathsf{v}_{14})(\mathsf{v}_{14} \longrightarrow \mathsf{v}_{15})(\mathsf{v}_{15} \longrightarrow \mathsf{v}_{16})\}$.

Overall, putting together the steps depicted in this section, we generate a set of sketch-specification pairs $(\mathsf{sk}, \mathsf{spec})$ that each represent a common editing sequence in the input Traces.

5 SYNTHESIZING EDIT SEQUENCE PATTERNS

From Section 4, we get as input a number of sketch-specification pairs. Here, we synthesize a hierarchy of ESPs for each sketch-specification pair. The procedure to do this has 3 major components: (a) generate an ESP from an edit sequence, (b) combine two ESPs to a more general pattern, and (c) produce a hierarchy of ESPs using the previous two components. Components (a), (b), and (c) are explained in Sections 5.1, 5.2, and 5.3, respectively.

5.1 Generating an Edit Sequence Pattern

Consider generating an ESP from an edit sequence $\mathsf{ed}_1 \dots \mathsf{ed}_n$ and a sketch $\mathsf{sk} = A_1 \dots A_n$.

- First, for each edit ed_i, let $\mathsf{Localize}(\mathsf{ed}_i) = \mathsf{ed}_{i,\mathsf{pre}} \longrightarrow \mathsf{ed}_{i,\mathsf{post}}$. We set $\mathsf{et}_i = \mathsf{ed}_{i,\mathsf{pre}} \longrightarrow \mathsf{ed}_{i,\mathsf{post}}$ and $\mathsf{Preds} = \emptyset$ and iteratively perform the following operations. (a) Identify AST nodes $\mathsf{node}_{\mathsf{pre}}$ in $\mathsf{et}_{i,\mathsf{pre}}$ and $\mathsf{node}_{\mathsf{post}}$ in $\mathsf{et}_{i,\mathsf{post}}$ such that there is a predicate that relates the two values. Further, we pick $\mathsf{node}_{\mathsf{pre}}$ and $\mathsf{node}_{\mathsf{post}}$ such that they are of the largest possible size. For example, we may pick $\mathsf{node}_{\mathsf{pre}}$ and $\mathsf{node}_{\mathsf{post}}$ such that $\mathsf{node}_{\mathsf{post}} = \mathsf{node}_{\mathsf{pre}}$ or $\mathsf{node}_{\mathsf{post}} = \mathsf{ToLower}(\mathsf{node}_{\mathsf{pre}})$. (b) We replace $\mathsf{node}_{\mathsf{pre}}$ and $\mathsf{node}_{\mathsf{post}}$ in et_i by two fresh holes $\mathsf{H}_{\mathsf{pre}}$ and $\mathsf{H}_{\mathsf{post}}$, and add the predicate that relates the two values to Preds (e.g., $\mathsf{H}_{\mathsf{post}} = \mathsf{H}_{\mathsf{pre}}$ or $\mathsf{H}_{\mathsf{post}} = \mathsf{ToLower}(\mathsf{H}_{\mathsf{pre}})$). We will also add unary predicates IsNotNull and IsKind of $\mathsf{H}_{\mathsf{post}}$ and $\mathsf{H}_{\mathsf{pre}}$ if they satisfy the constraints. (c) We add the generated mappings to a substitution σ_i.
- Then, we union all the substitutions σ_i into a single σ (note that domains of σ_i are disjoint). Now, for each $\mathsf{H}_1 \longrightarrow \mathsf{node}_1, \mathsf{H}_2 \longrightarrow \mathsf{node}_2 \in \sigma$, we check if there exists a predicate that relates node_1 and node_2. If so, we add that predicate on H_1 and H_2 to the Preds.

The produced ESP is $\langle \mathsf{et}_1 \dots \mathsf{et}_n, \mathsf{Preds} \rangle$. Note that the above procedure is for sketches without Kleene stars—we discuss how to generate patterns with Kleene stars later.

Example 5.1. Consider the edit sequence $\mathsf{ed}_1\mathsf{ed}_2\mathsf{ed}_3$ where $\mathsf{ed}_1 = \mathsf{v}_6 \longrightarrow \mathsf{v}_8$, $\mathsf{ed}_2 = \mathsf{v}_8 \longrightarrow \mathsf{v}_9$, and $\mathsf{ed}_3 = \mathsf{v}_9 \longrightarrow \mathsf{v}_{10}$ from Figure 3. We illustrate the ESP generation procedure for ed_1. First, we start with the localized edit $\mathsf{Localize}(\mathsf{ed}_1)$ = `{<IdDecl> <Ctor>}` ⟶ `{<IdDecl> public int Size {get; set;} <Ctor>}`. The terms `<IdDecl>` and `<Ctor>` are shorthand for `public int Id {get; set;}` and `Graph(int id) {Id = id;}`, respectively.

Now, over all pairs of nodes in the localized edit, we check if there is a predicate that is satisfied by the nodes. Here, we get that the node `<IdDecl>` is repeated in both the pre- and post-versions. Replacing these with holes, we get the edit template `{`H_1 `<Ctor>}` $\longrightarrow$ `{`H_3 `public int Size {get; set;} <Ctor>}` and the predicate $\mathsf{H}_1 = \mathsf{H}_3$. Repeating this, we replace `<Ctor>` with H_2 and H_4 to get the edit template et_1 = `{`H_1 H_2`}` $\longrightarrow$ `{`H_3 `public int Size {get; set;}` H_4`}` and the predicate $\mathsf{H}_4 = \mathsf{H}_2$.

Doing the similar procedure on ed_2 and ed_3, we get the edit templates et_2 = `(`H_5`)` $\longrightarrow$ `(`H_6`, int size)` and et_3 = `{`H_7`}` $\longrightarrow$ `{`H_8 `Size = size;}`, with the predicates $\mathsf{H}_6 = \mathsf{H}_5$ and $\mathsf{H}_8 = \mathsf{H}_7$. We then compute the predicates across the different et_i, and in this example, we do not find any.

The ESP returned is $\mathsf{et}_1\mathsf{et}_2\mathsf{et}_3$ along with the predicates $\mathsf{Preds} = \{\mathsf{H}_3 = \mathsf{H}_1, \mathsf{H}_4 = \mathsf{H}_2, \mathsf{H}_6 = \mathsf{H}_5, \mathsf{H}_8 = \mathsf{H}_7\}$. Note that the pattern does not create holes for the type or name of the inserted property or parameter (`int`, `Size`, and `size`). With just a single edit sequence, we do not have any evidence for the need to generalize these identifiers—hypothetically, every property that is added in the input traces might have the type `int` and name `Size`. We can generalize these identifiers when we have two ESPs as we will describe next. □

5.2 Combining Edit Sequence Patterns

Using the previous step, we can generalize all concrete edit sequences in spec to ESPs. Now, we discuss how to combine any two such ESPs into a single, more general, ESP. The primary tool we use for this purpose is *anti-unification* [Plotkin 1970]. Anti-unification is a classical operation of trees that retains the parts that are common to two trees, while replacing the parts that are different with holes. It has been used in code edit analysis and synthesis literature to generalize ASTs and edits in multiple contexts [Bader et al. 2019; de Sousa et al. 2021; Gao et al. 2020]. However, in our technique, we need to anti-unify sequences of edit templates rather than ASTs or single edit templates, and further, need to consider the predicates.

Formally, given two sequences of edit templates $\mathsf{et}_1 \dots \mathsf{et}_n$ and $\mathsf{et}'_1 \dots \mathsf{et}'_n$, the anti-unification of the two sequences produces an edit template sequence $\mathsf{et}^*_1 \dots \mathsf{et}^*_n$ and two substitutions σ, σ' such that: For each et^*_i, we have that $\mathsf{et}_i = \sigma(\mathsf{et}^*_{i,\mathsf{pre}}) \longrightarrow \sigma(\mathsf{et}^*_{i,\mathsf{post}})$ and $\mathsf{et}'_i = \sigma'(\mathsf{et}^*_{i,\mathsf{pre}}) \longrightarrow \sigma'(\mathsf{et}^*_{i,\mathsf{post}})$. Intuitively, anti-unification is generalization: if any edit sequence $\mathsf{ed}_1 \dots \mathsf{ed}_n$ matches $\mathsf{et}_1 \dots \mathsf{et}_n$ or $\mathsf{et}'_1 \dots \mathsf{et}'_n$, it will also match $\mathsf{et}^*_1 \dots \mathsf{et}^*_n$. Further, we also need to generalize hole predicates Preds and Preds'. For this, we generate only those hole predicates that are satisfied by both substitutions σ and σ'. As a result, the newly generated hole predicates is also a generalization.

Example 5.2 (Anti-unification of edit sequence patterns). Recall the edit templates $\mathsf{et}_1\mathsf{et}_2\mathsf{et}_3$ from Example 5.1. Now consider another ESP from a similar sequence of edits shown in Figure 2, but with the following changes: (a) the property and parameter added has a different name and type (`str Id` and `str id`) and (b) the parameter list in the constructor and the body of the constructor are empty. In the edit templates $\mathsf{et}'_1\mathsf{et}'_2\mathsf{et}'_3$ for an ESP generated for this case, et'_1 is similar to et_1 with the hole names replaced. However, et'_2 = `()` $\longrightarrow$ `(str id)`, i.e., it does not have H_5 and H_6 to represent the already existing parameters in the parameter list. Similarly, et'_3 = `{ }` $\longrightarrow$ `{Id = id;}` and it does not contain H_7 and H_8.

To generalize the two edit templates $\mathsf{et}_1\mathsf{et}_2\mathsf{et}_3$ and $\mathsf{et}'_1\mathsf{et}'_2\mathsf{et}'_3$, we anti-unify the before and post templates in each et_i and et'_i one by one:

- For et_1 and et'_1, we get the anti-unified edit template et^*_1 = `{`H^*_1 H^*_2`}` $\longrightarrow$ `{`H^*_3 `public` H^*_9 H^*_{10} `{get; set;}` H^*_4`}`. Note that the type and name of the properties have been replaced by new holes H^*_9 and H^*_{10}.
- For generalizing, et_2 and et'_2, additional care must be taken as there are no holes corresponding to H_5 and H_6 in et'_2. Some anti-unification approaches [Bader et al. 2019; de Sousa et al. 2021]

  will produce an overly general edit template et_2^* as $(\mathsf{H}_{11}^*) \longrightarrow (\mathsf{H}_{12}^*)$. With this edit template, we do not have any holes for the name and type of the parameter, and thus we cannot express the hole predicate between parameter type and property type.

  We propose to further generalize the lists of children, i.e., (H_6, `int size`) and (`str id`), inspired by Gao et al. [2020]. During anti-unification, we examine if the two children lists can be better generalized by introducing additional holes which are substituted by the empty token ϵ in one case. Doing so, we get $\mathrm{et}_2^* = (\mathsf{H}_5^*) \longrightarrow (\mathsf{H}_6^*,\ \mathsf{H}_{11}^*\ \ \mathsf{H}_{12}^*)$.
- Similarly, for et_3 and et_3', we get $\mathrm{et}_3^* = \{\mathsf{H}_7^*\} \longrightarrow \{\mathsf{H}_8^*\ \ \mathsf{H}_{13}^* = \mathsf{H}_{14}^*\}$.

Note that we can get substitutions σ and σ' for free after we generalized these edit templates. These substitutions can then be used to generate the hole predicates. (a) For a specific unary hole predicate F, we enumerate every hole H in generalized edit templates and checking whether both $\mathsf{F}(\sigma(\mathsf{H}))$ and $\mathsf{F}(\sigma'(\mathsf{H}))$ are satisfied. Notice that $\sigma(\mathsf{H})$ and $\sigma'(\mathsf{H})$ can still contain holes in the first or the second ESP. If there are any holes in $\sigma(\mathsf{H})$ or $\sigma'(\mathsf{H})$, we recursively repeat the substitution procedure until H maps to an AST in concrete edit sequences. (b) We generate binary hole predicates similarly but enumerate all pairs of holes in generalized edit templates. In this case, we end up with the starred version of the predicates in Preds from Example 5.1, along with the predicates $\{\mathsf{H}_{11}^* = \mathsf{H}_9^*, \mathsf{H}_{12}^* = \mathsf{ToLower}(\mathsf{H}_{10}^*), \mathsf{H}_{13}^* = \mathsf{H}_{10}^*, \mathsf{H}_{14}^* = \mathsf{H}_{12}^*\}$.

The generalized edit templates $\mathrm{et}_1^*\mathrm{et}_2^*\mathrm{et}_3^*$ along with the new predicates exactly capture the editing sequence of adding a new property and initializing it in the constructor. □

As illustrated in the previous example, we cannot generalize ESPs using standard anti-unification techniques. When two nodes in the edit templates have different numbers of children, we may need to introduce new holes that map to ϵ. We do not explicitly write out our anti-unification algorithm here for the lack of space—instead, it is available in the supplementary material. The algorithm takes as input two ESPs $\langle \mathsf{TS}_1, \mathsf{Preds}_1 \rangle$ and $\langle \mathsf{TS}_2, \mathsf{Preds}_2 \rangle$, and produces a more general $\langle \mathsf{TS}, \mathsf{Preds} \rangle$. Along with the generalized ESP, the algorithm also returns a cost of anti-unification. This cost roughly measures how general the $\langle \mathsf{TS}, \mathsf{Preds} \rangle$ is compared to $\langle \mathsf{TS}_1, \mathsf{Preds}_1 \rangle$ and $\langle \mathsf{TS}_2, \mathsf{Preds}_2 \rangle$, with more general patterns getting a higher cost than less general ones. This corresponds to the intuition that anti-unification algorithms attempt to compute the least general generalization of two objects. Based on the anti-unification costs, we will generate a hierarchy of ESP in the following section.

5.3 Building a Hierarchy of Edit Sequence Patterns

Algorithm 2 The procedure of building a dendrogram (LearnPatterns in Algorithm 1)

```
Require: Sketch of edit sequence pattern sk
Require: Specification for edit sequence pattern spec
Ensure: A set of edit sequence patterns Patterns
Nodes ← {GeneratePattern(sk, $\mathrm{ed}_1 \ldots \mathrm{ed}_n$) | $\mathrm{ed}_1 \ldots \mathrm{ed}_n \in$ spec}
Patterns ← ∅
while |Nodes| > 1 do
    Pick $\mathsf{Node}_1, \mathsf{Node}_2$ such that AntiUnifyCost($\mathsf{Node}_1, \mathsf{Node}_2$) is minimal
    NewNode ← AntiUnify($\mathsf{Node}_1, \mathsf{Node}_2$)
    Patterns ← Patterns ∪ {NewNode}
    Nodes ← Nodes − {$\mathsf{Node}_1, \mathsf{Node}_2$} ∪ {NewNode}
return Patterns
```

Algorithm 2 shows the full procedure going from an ESP sketch sk and specification spec to a set of ESPs Patterns. The algorithm performs a standard agglomerative hierarchical clustering (AHC) [Day and Edelsbrunner 1984] with the distance metric given by the anti-unification cost.

AHC builds a dendrogram where each node is an ESP. This is reminiscent of the techniques [Bader et al. 2019], but performed over a sequence of edits rather than a single one. At line 1, we build the leaf nodes in dendrogram by generalizing edit sequences to ESPs as described in Section 5.1. At lines 4-5, we select two nodes in the dendrogram $\mathsf{Node}_1, \mathsf{Node}_2$ that have the lowest merging anti-unification cost in anti-unification and anti-unify them into a new dendrogram node NewNode. The procedure eventually returns the set of all nodes that were constructed.

Remark 5.3 (Handling Kleene stars). An ESP sketch $A_1 \dots A_{n-1}A_n^{[*]}$ with Kleene stars may contain edit sequences with various lengths. To handle the sketch with Kleene stars, we will compute a new set of edit sequences, each of which has length n equal to the sketch, such that we can build the dendrogram in the same way as the sketches without Kleene stars. Concretely, for an edit sequence $\mathsf{ed}_1^i \dots \mathsf{ed}_m^i$ in the sketch we will collect all subsequences $\mathsf{ed}_1^i \dots \mathsf{ed}_{n-1}^i \mathsf{ed}_k^i$ for all $n \leq k \leq m$.

Suppose we have an ESP sketch $A_1A_2^*$ and an edit sequence $\mathsf{ed}_1\mathsf{ed}_2\mathsf{ed}_3\mathsf{ed}_4$, where ed_1 corresponds to A_1 and $\mathsf{ed}_2, \mathsf{ed}_3, \mathsf{ed}_4$ correspond to A_2. We will break the edit sequence into three edit sequence with length 2, namely, $\mathsf{ed}_1\mathsf{ed}_2$, $\mathsf{ed}_1\mathsf{ed}_3$, and $\mathsf{ed}_1\mathsf{ed}_4$.

6 RANKING EDIT SEQUENCE PATTERNS

As we discussed in Section 2, we cannot simply pick more general ESPs over less general ones. More general patterns may be less predictive than specific ones. In this section, we will select a ranked list of ESPs from all dendrograms as the output of Overwatch.

Algorithm 3 The procedure of edit sequence pattern prediction

```
Require: An edit sequence pattern $\langle \mathsf{TS} = \mathsf{et}_1 \dots \mathsf{et}_{n-1}\mathsf{et}_n^{[*]}, \mathsf{Preds} \rangle$
Require: A trace $\mathsf{v}_0 \dots \mathsf{v}_m$
Ensure: Prediction for next version $\hat{\mathsf{v}}$ or $\bot$
 1: for all possible edit sequences $\mathsf{ed}_1 \rightarrow_{\mathsf{seq}} \dots \rightarrow_{\mathsf{seq}} \mathsf{ed}_k$ ending at $\mathsf{v}_m$ and $k \leq n$ do
 2:     if $\mathsf{ed}_1 \dots \mathsf{ed}_k$ matches $\mathsf{et}_1 \dots \mathsf{et}_k$ using a unique $\sigma$ then
 3:         if $k < n$ then
 4:             $\mathsf{v}_{\mathsf{predicted}} \leftarrow \textsc{Predict}(\mathsf{et}_{k+1}, \sigma)$
 5:             if $\mathsf{v}_{\mathsf{predicted}} \neq \bot$ then return $\mathsf{v}_{\mathsf{predicted}}$
 6:         else
 7:             $\mathsf{v}_{\mathsf{predicted}} \leftarrow \textsc{Predict}(\mathsf{et}_n, \sigma)$
 8:             if $\mathsf{v}_{\mathsf{predicted}} \neq \bot$ then return $\mathsf{v}_{\mathsf{predicted}}$
 9: return $\bot$
10: function Predict($\mathsf{et} = \mathsf{t}_{\mathsf{pre}} \rightarrow \mathsf{t}_{\mathsf{post}}, \sigma$)
11:     for all Every subtree $\mathsf{v}_m^*$ in $\mathsf{v}_m$ do
12:         if $\exists \sigma'.\sigma'(\mathsf{t}_{\mathsf{pre}}) = \mathsf{v}_m^* \wedge \sigma'$ satisfy Preds $\wedge\ \sigma \subseteq \sigma'$ then
13:             return $\mathsf{v}_m$ with the subtree $\mathsf{v}_m^*$ replaced by $\sigma'(\mathsf{t}_{\mathsf{post}})$
14:     return $\bot$
```

Predictions Using Edit Sequence Patterns First, we discuss how an ESP can be used for predicting the next change in Algorithm 3. As input, it takes an ESP $\langle \mathsf{TS}, \mathsf{Preds} \rangle$ and a trace $\mathsf{v}_0 \dots \mathsf{v}_m$. Algorithm 3 will first match a developer's edits against a prefix of a learned ESP (Lines 1-2) and use the next edit template in the ESP to predict the change the developer is going to make next (Lines 3-8).

Remark 6.1. In our implementation, instead of brute-force enumeration at Lines 1-2, we enumerate all matched edit sequences by prefixes of an ESP in polynomial time. We do this by matching edits on a deterministic finite automaton generated from the ESP. Also, note that we require the substitution σ to be *unique* to avoid over-generalization. Notice that we only let an ESP generate its first prediction for a trace in Algorithm 3. However, we recorded all predictions that can be made by an ESP in our evaluation and found that no ESP had made multiple predictions for a trace.

Remark 6.2. In our evaluation, we provide the cursor location information to `Predict` in Algorithm 3 such that we only need to enumerate subtrees v^*_m that contain the cursor location of the user to make more precise predictions. We predict one single edit at a time because this evaluation methodology precisely corresponds to how the ESPs are deployed. In practice, the fact that the user navigated the cursor to the location for the next edit is (a) a confirmation that the prediction is likely to be correct and (b) ensures that we do not interrupt the user's workflow.

For each ESP, there are three outcomes at each version v_m: (1) the pattern does not predict, i.e., returns $\perp$, (2) *correct* prediction, the pattern predicts $\hat{\mathsf{v}}$ that is equal to v_l, $l > m$ in the later sessions, and (3) otherwise, we consider the pattern makes a *wrong* prediction.

Ranking and Selecting Edit Sequence Patterns. From Section 5, we get a set of ESPs Patterns. Let edSeqs be the union of all edit sequences in the specifications generated in Section 4. Patterns may contain noisy ESPs that have high precision on the edit sequences in its own specification, but low precision when evaluated on all version data. Thus, we try to solve the following problem: select and rank a subset of Patterns such that we maximize the correctly predicted versions and minimize the wrongly predicted versions. Algorithm 4 depicts the procedure for this.

Algorithm 4 The procedure of ranking edit sequence patterns (FilterAndSelect in Algorithm 1)

```
Require: A list of edit sequence patterns Patterns
Require: Input traces Traces
Require: Edit sequences EditSeqs
Require: Precision thresholds $\mathsf{threshold}_1, \mathsf{threshold}_2$
Ensure: A ranked list of edit sequence patterns Patterns
function FilterAndSelect
    Patterns ⟵ GreedySelect(Patterns, EditSeqs, $\mathsf{threshold}_1$)
    Patterns ⟵ GreedySelect(Patterns, Traces, $\mathsf{threshold}_2$)
    return Patterns
function GreedySelect(Patterns, Data, threshold)
    SelectPatterns ⟵ [], Uncovered ⟵ Data
    for all $p_i \in$ Patterns do
        $\mathsf{correct_i}, \mathsf{incorrect_i}$ ⟵ Evaluate($p_i$, Data)
    Patterns ⟵ $\{p_i \mid \frac{\mathsf{correct_i}}{\mathsf{correct_i}+\mathsf{incorrect_i}} > \mathsf{threshold}\}$
    while Patterns ≠ ∅ ∧ Uncovered ≠ ∅ do
        $p$ ⟵ $\mathrm{argmax}_{p_i \in \mathsf{Patterns}} \mathsf{correct_i} - \mathsf{incorrect_i}$
        Patterns ⟵ Patterns − $\{p\}$
        SelectPatterns ⟵ SelectPatterns + $[p]$
        Uncovered ⟵ Uncovered − {data points covered by $p$}
        Update all $\mathsf{correct_i}$ and $\mathsf{incorrect_i}$ according to Uncovered
    return SelectPatterns
```

The core component of Algorithm 4 is the GreedySelect procedure. The procedure takes as input a set of patterns Patterns and traces Data and produces a ranked subset of patterns based on their predictive performance. Intuitively, GreedySelect works similar to the approximate set-cover algorithm. We call each version in a trace in Data a data point. The procedure maintains a partial list of selected patterns and a set Uncovered of data points on which no selected pattern has made a prediction. We first measure the number of correct and incorrect predictions each pattern makes on Data. With these count of correct and incorrect predictions, we then eliminate all patterns with precision less than a given threshold. In each iteration of selection, (a) we update the partial list of pattern with the best pattern as measured by the difference in the number of correct and incorrect predictions, and (b) we remove the set of datapoints on which the best pattern made predictions from Uncovered and update all $\mathsf{correct}_i$ and $\mathsf{incorrect}_i$ accordingly.

Algorithm 4 calls GreedySelect twice in two phases. In the first, we only consider the predictive power of each pattern on the set edSeqs. Then, in the second step, we select and rank based on the full training data, i.e., the input Traces. Ideally, we only need the second step at line 3 because the precision on traces reflects the effectiveness of ESPs in real scenarios. However, we add the first step of filtering to reduce the number of patterns evaluated in the second step because performing GreedySelect over traces on all patterns is very expensive. We find the first step is able to filter out most of the patterns, speeding up the second step by a large degree.

7 EVALUATION

In this section, we present our evaluation to address the following research questions:

- **RQ1**: How effective is Overwatch at predicting edit sequences performed in the IDE?
- **RQ2**: What kind of ESPs are learned by Overwatch?
- **RQ3**: How do different components of Overwatch compare to state-of-the-art techniques?

7.1 Data Collection

We developed a Visual Studio extension to record all syntactically correct versions of the documents updated by the developer (in the background, without interruption). We selected Visual Studio as the target IDE in this study as it is the most popular IDE for C#. We contacted **12** professional software developers from a large software company who agreed to use the extension and participate in the study. They were working on **four** separate C# code bases with a total of 377.5K source lines of code. Initially, we recorded **682** *development sessions* (≅ 250 hours) containing **134, 545** *versions* of **425** *documents*, which we refer to as *training dataset*. After 6 months, we collected an additional **399** sessions (containing **201, 142** versions), which we refer to as *test dataset*.

7.2 RQ1: Effectiveness of Overwatch

7.2.1 Experimental Setup. Overwatch requires a few runtime parameters : (a) The maximum length n of the sequences, (b) minimum support for the edit sequences s (Section 4), (c) thresholds $\mathsf{threshold}_1$ and $\mathsf{threshold}_2$ for selecting patterns based on the sketch and session analysis, respectively (Section 5). We found $\mathsf{n} = 3$ to be the sweet spot for learning useful patterns and chose $\mathsf{s} = 2$ for the support so that we have at least two examples for each pattern. For larger n, Overwatch will generate overly general patterns with unbound holes, i.e., holes for which values cannot be predicted using the predicates learned. We study the effect of varying the values of $\mathsf{threshold}_1$ and $\mathsf{threshold}_2$ in the experiments.

To answer RQ1, we (a) perform a 5-fold validation over our *training dataset* and (b) simulate the learned patterns from the training dataset on our *test dataset* containing unseen development sessions. To perform the 5-fold validation, we randomly split **682** training sessions into 5 equal *folds* and for each fold, we evaluate the ESPs that were learned from the other 4 folds. We repeat the 5-fold validation varying $\mathsf{threshold}_1, \mathsf{threshold}_2 \in [0, 1]$ by steps of 0.1. Using the best threshold values found (as measured by the F_3 metric defined below), we evaluate the ESPs learned from the full training dataset on the test dataset.

For each ESP and version v in the dataset, we follow Algorithm 3 to predict the next version, i.e., to produce a code edit suggestion. We compute the *precision* of the code edit suggestions as the proportion of the total suggestions that are *correct* (see Section 6). To compute *recall*, we

Table 1. Precision and relative recall of Overwatch sorted by 5-fold Validation F_3 scores

		5-fold Validation (average)			**Test Set Evaluation**			
Threshold$_1$	Threshold$_2$	Precision (in %)	Recall$_{rel}$ (in %)	F$_3$ (in %)	Precision (in %)	Recall$_{rel}$ (in %)	F$_3$ (in %)	#Correct
0.7	0.8	79.47	40.73	72.57	78.38	36.25	70.22	145
0.6	0.7	76.07	49.75	72.25	70.93	40.25	65.90	161
0.6	0.8	78.47	41.46	72.04	77.42	36.00	70.22	144
0.8	0.8	78.38	41.35	71.94	82.65	40.50	74.86	162
0.5	0.8	78.15	40.08	71.37	74.29	32.50	65.82	130
Baseline (0, 0)		49.23	100	51.86	38.17	100	40.68	400

need an oracle containing all the predictions expected from Overwatch. However, these expected predictions (ground truth) are not easy to generate automatically. Overwatch is designed to infer ESPs corresponding to patterns in developer's editing behaviour—some of these patterns are not known and do not correspond to any known refactoring or automated tool in any IDE. Furthermore, it is not feasible to construct such an oracle by manual annotation because of scale—our dataset consists of hundreds of thousands of fine-grained edits. Instead, for our experiments, we define our baseline as the number of correct suggestions we get when using Overwatch in its most general setting with threshold$_1$ = 0 and threshold$_2$ = 0 values in Algorithm 4. Using this baseline, we define relative recall (*recall$_{rel}$*) as the ratio of correct predictions Overwatch produces at a given configuration with respect to the baseline. For example, if the baseline makes 400 correct predictions and Overwatch with a new configuration threshold$_1$ = 0.7 and threshold$_2$ = 0.8 makes 145 correct predictions, the relative recall of Overwatch with the new configuration is Recall$_{rel}$ = 145/400 = 36.25%.

To consolidate the *precision* and *recall* metric into a single score, we make use of a variation of the popular F_β metric [Chinchor 1992; van Rijsbergen 1979]. Here, we use F_γ given by:

$$F_\gamma = \frac{(1+\gamma^2) * precision * recall_{rel}}{precision + \gamma^2 recall_{rel}} \tag{1}$$

Note that this definition is equivalent to the definition of F_β with β set to $\frac{1}{\gamma}$ and the recall term replaced by relative recall. The γ parameter allows us to choose the relative emphasis we put on the *precision* term compared to *recall* term, with *precision* given γ times more importance over *recall* [van Rijsbergen 1979]. For our evaluation, we make use of $F_{\gamma=3}$ to give *precision* 3 times more importance than *recall$_{rel}$*. We chose to increase the emphasis on precision because of two reasons: (a) Reliability is one of main causes of disuse of automated refactorings in IDEs [Vakilian et al. 2012], so tool builders tend to favor precision over recall. (b) The use of *recall$_{rel}$* instead of *recall* tend to introduce a bias towards recall because *recall$_{rel}$* will be higher than ground-truth *recall.*

For notational simplicity we use F_3 instead of $F_{\gamma=3}$ to refer to this metric in the rest of the paper.

7.2.2 *Results.*

Table 1 summarizes the precision, relative recall and F_3 statistics for the top-5 threshold configurations for both the 5-fold validation and the test set evaluation. In the bottom row, we also report the statistics for the *baseline* configuration used to calculate relative recall. The baseline configuration made 1048 predictions over the full test set out of which 400 were correct. Among the top-5 configurations, Overwatch's precision on the test set ranged from 70.93% to 82.65% compared to baseline's 38.17%, their relative recall ranged from 32.25% to 40.25% compared to Baseline's 100%, and their F_3 ranged from 65.82% to 74.86% compared to Baseline's 40.68%. The best configuration on 5-fold validation set uses threshold$_1$ = 0.7 and threshold$_2$ = 0.8 and achieves 78.38% precision, 36.25% relative recall, and 70.22% F_3 on the test set evaluation. We also present the number of correct predictions on the test set in the last column of Table 1.

Comparing the statistics for 5-fold validation and test set evaluation, we observe high parallels in terms of the precision and relative recall, hinting towards a degree of *domain-invariance* in the learned patterns as the train dataset and test dataset were collected more than 6 months apart.

The effectiveness of Overwatch has a degree of domain-invariance and the best configuration on 5-fold validation achieves 78.38% precision, 36.25% relative recall, and 70.22% F_3 on the test set evaluation.

7.3 RQ2: Nature of learned ESPs

7.3.1 Experimental Setup. To answer RQ2, we manually analyzed the ESPs learned by Overwatch using the best threshold parameters found from the RQ1 study. Two authors, each with more than 5 years of professional development in C#, coded these ESPs using established guidelines from the literature [Campbell et al. 2013; Saldana 2009]. They first iteratively refined the code set on 20% of the patterns. Then, using this code set, they independently coded another 20% of the patterns. We then use Cohen's kappa to calculate inter-rater reliability. Their inter-rater reliability was 0.95, which shows a high agreement between the two raters. We then split the rest of the patterns into two sets, and they independently coded each set. We detail each code set in RQ2.

7.3.2 Results. Next, we present the results of our qualitative analysis.
Categorizing ESPs. Fixing the best configuration ($\text{threshold}_1 = 0.7$ and $\text{threshold}_2 = 0.8$) from the previous study, Overwatch learned 135 edit sequence patterns. We classified these patterns into four categories using the coding methodology from Section 7.3.1, as shown in Table 2.

- *Workflows.* We classified 25.9% of the ESP as *Workflow*, which consists of an ESP that describe the workflow of a developer performing a particular high-level task. For instance, to rename a variable, the developer first renames the variable in the declaration, and then renames each one of the variable uses. Each step of the workflow is represented as an edit template in the ESP. The ESP detects the step that the developer is in the task, and predicts the next steps to finish it.
- *Repeats.* 27.5% of the ESP were classified as *Repeat*, which consists of ESP that represent a developer performing a single task multiple times. For instance, a developer performs an edit to remove the qualifier "this" from multiple parts of the code. The ESP detects that the developer performed the edit in one location and when they move to another similar location, the ESP predicts the change.
- *Transients and Noise.* Finally, we identified two categories of ESP that are not useful: *Transient* and *Noise*. The former (13%) relates to edits that are too fine-grained, such as inserting `publicclass` and then changing to `public class`. The latter (33.6%) relates to changes that do not represent a high-level task, such as adding a specific switch statement and then adding a break statement.

After removing noisy ESPs and accounting for ESPs that are variations of a single type of refactoring, we get 51 unique pattern types. The list of these 51 pattern types is shown in supplementary material.
Relating ESPs to IDE features. We further sub-classified the Workflow and Repeat ESPs into two categories: *Existing Feature* and *New Feature*. Table 3 presents a list of 20 learned ESPs, 10 existing features and 10 new features. In the first category, we include ESPs that have a corresponding automated tool or refactoring implemented in the IDE, which allows the IDE to automate the complete edit in one step using just the spatial context. Existing Feature ESPs correspond to 53% and 39% of the Workflow and Repeat ESPs, respectively. The New Feature category includes ESPs for which we did not find a corresponding IDE feature. For instance, ESP 9 is the inverse of ESP 6, Instead of adding a property and its corresponding parameter, the developers removes the property

Table 2. Categories of Edit Sequence Patterns

Id	Category	Description	Examples	%
1	Workflow	A pattern that represents the workflow that a developer performs to complete a task. Each edit in the edit sequence represent a step that the developer took. The temporal context is used to detect the previous steps that the developer performed and predict the remaining ones.	(i) Developer changes the name of a variable in its declaration and then renames each one of the references (rename variable); (ii) Developer changes the type of a variable in the left-hand side of an assignment and then changes the name of the constructor in the right-hand side of the assignment.	25.9
2	Repeat	A pattern that represents a developer performing the same task multiple times. All edits in the sequence have the same edit template. The temporal context predicts that the developer will perform the task again.	(i) A developer deletes the keyword `this` from multiple locations in a class (Remove this); (ii) Developer replaces a static method invocation with a virtual method invocation in multiple locations.	27.5
3	Transient	The sequence represented is too fine-grained to be considered useful.	(i) Developer inserts i, then changes to i++; (ii) Developer writes "publicclass" then changes to "public class"	13.0
4	Noise	We could not identify a high-level task for this pattern.	(i) Developer creates a switch statement with a specific case, and then adds a break statement; (ii) developer cuts and pastes a statement.	33.6

and the parameter. Note that this pattern shows that developers perform changes in a non-standard way—the ESP first deletes the parameter, then the assignment, and finally the property.

Analyzing Existing Feature ESPs. We discuss existing feature ESPs in detail here as they are closely connected to our motivation of addressing the late-awareness and discoverability problems. Overwatch learns existing feature ESPs only because developers manually performed these edits, which created a trace of fine-grained edits, instead of using the IDE tool support, which would lead to a single, larger edit. For instance, ESP 4 represents the edit sequence shown in Figure 7. The complete edit is automated in one step by Visual Studio (Delete Parameter) using the spatial context. To apply this refactoring, the developer needs to put the cursor on the parameter list, then click on the *Quick Actions* pop-up (the screwdriver icon on the left side of the text pane) select "Change signature..." among all the code edit options, and then select the parameter to delete. To apply this refactoring, not only does the developer need to be aware of this tool, but also needs to use the tool before making any changes manually. If the developer starts by deleting the parameter from the parameter list, Visual Studio will not generate a suggestion to finish the edit sequence by deleting the corresponding arguments. Meanwhile, ESP 4 uses the fact that the developer manually deleted the parameter to predict that the developer will delete the corresponding argument–the developer can use a tool based on ESP 4 even if they have already started making changes.

These results suggest that IDE features were under used, in congruence with the observation made by Ge et al. [2012]. They point to the fact that these tools are hard to discover (discoverability issue) and even when they are discoverable, developers do not realize the possibility of using it at the time when that suggestion is available (late-awareness). Overwatch can alleviate these problems by producing code edit suggestions using the learned ESPs as shown in Figure 1b, while the developer is editing (see the discussion section).

Our qualitative analysis shows that ESPs can be used not only to complete edits when developers typically miss the opportunity to use the IDE tool support but also to predict edits based on new patterns that have no tool support at all.

7.4 RQ3: Comparison to state-of-the-art

To our knowledge, there is no other technique or tool that addresses the problem of learning ESPs. However, we compare Overwatch to two related state-of-the-art techniques: C^{3}PO [Brody et al. 2020] and Blue-Pencil [Miltner et al. 2019].

Table 3. Sample of 20 Edit Sequence Patterns Learned by Overwatch

Category	Id	Pattern Description	Related Feature
Workflow	1	Rename method decl ⟶ Rename method calls	*Rename Method*
	2	Insert variable decl ⟶ Replace constants with new variable	*Introduce Local Variable*
	3	Insert parameter ⟶ Insert argument to callsites	*Insert parameter*
	4	Delete parameter ⟶ Delete argument from callsites	*Delete parameter*
	5	Replace variable declaration by assignment ⟶ Insert new field	*Promote local variable to field*
	6	Insert Property ⟶ Insert Parameter ⟶ Insert Assignment	*Initialize property*
	7	Insert expression ⟶ Replace it by assignment	*Introduce variable*
	8	Change type in variable decl ⟶ Change constructor name in initializer	*New Feature*
	9	Delete parameter ⟶ Delete assignment ⟶ Delete property	*New Feature*
	10	Insert parameter with default value ⟶ Replace constants with parameter	*New Feature*
	11	Delete field ⟶ Delete assignment	*New Feature*
	12	Insert argument to callsite ⟶ Remove default parameter value	*New Feature*
	13	Insert variable declaration ⟶ Insert new variable as argument	*New Feature*
	14	Insert return statement ⟶ Delete throw "NotImplementedException"	*New Feature*
Repeat	15	Remove ‘this’ in multiple locations	*Remove unnecessary qualifier “this”*
	16	Remove ‘cast’ in multiple locations	*Remove unnecessary cast*
	17	Change from qualified name to simple name	*Simplify Name Access*
	18	Converting static method calls to virtual in multiple locations	*New Feature*
	19	Remove a method invocation from many locations	*New Feature*
	20	Replace an expression by a method invocation	*New Feature*

7.4.1 Comparison with C^{3}PO. Recently researchers Brody et al. [2020] addressed the problem of predicting the (next) edit that could be applied to a code snippet, given a previously applied edit to *the same code snippet*. As input, C^{3}PO requires a code snippet and a series of edits as input. On the other hand, Overwatch takes as input a code snippet and is a series of source file version that were produced during a development session in a IDE. As a sub-goal, Overwatch summarizes the series of source code files into edit sequences of appropriate granularity. Moreover the series of edits that C^{3}PO accepts as input, have to be such that they were applied in spatially proximity of the location where the prediction is made. On the other hand, Overwatch has no such limitation (as seen from our motivating example - Figure 4).

While the Overwatch and C^{3}PO are not directly comparable, it would be interesting to evaluate how C^{3}PO performs on our dataset of fine grained edit sequences. This would give us insight into the applicability of such deep learning techniques at edit sequence completion tasks in an IDE setting. Therefore, we evaluate C^{3}PO's effectiveness at predicting the next edit in the edit sequences generated in the sketch-and-specification generation step of Overwatch (Algorithm 1 - line 6). From each edit sequence $\mathrm{ed}_1 \ldots \mathrm{ed}_n \mathrm{ed}_{n+1}$ in a specification, we can create an input to C^{3}PO using the location of the edit ed_{n+1} as the code snippet and $\mathrm{ed}_1 \ldots \mathrm{ed}_n$ as the edits, and test if C^{3}PO can predict ed_{n+1}. Note that this is unlike the original setup for C^{3}PO, as $\mathrm{ed}_1 \ldots \mathrm{ed}_n$ are on different locations than the location of the code snippet where the prediction is to be made.

Dataset. Running the sketch-and-specification generation on the training and test datasets shown in Section 7.1, we end up with 9958 and 13532 edit sequences—we call these the *edit sequence training and test datasets*, respectively.

Experimental Setup. To perform this evaluation, we used open source codebase of C^{3}PO, with suitable adjustments for ingesting overwatch data[1]. For training and evaluation, we use the exact scripts provided by the C^{3}PO authors. In the first step, the script preprocesses the dataset to remove edit sequences that are not supported by the model. In our dataset, C^{3}PO cannot handle most of the edit sequences of the edit sequence training and test sets. Most importantly, it cannot deal with *generative insertions*, i.e., edits where new code that is not present in the input code snippet is inserted (82% of the training data). This is because the edit representation that C^{3}PO uses internally to create edit predictions does not account for such changes. For example, it cannot handle edits related to the ESP *Insert-Property-Parameter-Assignment* from Section 2. Further, C^{3}PO is also limited by: (a) the size of the edits (originally edits of at most 50 AST nodes, which we increased to 100); (b) other edits not being expressible in their edit representation in [Brody et al. 2020]; and (c) in a small fraction of the cases, C^{3}PO's parser rejecting valid C# code that the official C# parser accepts[2]. After filtering out all edit sequences containing edits that C^{3}PO cannot handle, we are left with only 453 and 314 edit sequences in the training and test edit sequence sets, respectively.

We ran experiments with 3 different configurations of C^{3}PO: (1) C^{3}PO(a) trained on the original training data from [Brody et al. 2020]; (2) C^{3}PO(b) trained on the training edit sequence data; (3) C^{3}PO(c) trained on the original training data from [Brody et al. 2020] and fine-tuned using the training edit sequence data.

Results. Among the C^{3}PO configurations, C^{3}PO(c) performs the best, producing correct predictions on 32 of the 314 test sequences, giving us an accuracy of 10.5%. C^{3}PO(a) and C^{3}PO(b) have an accuracy of 0.3% and 7.3%, respectively.

The results for C^{3}PO(a) suggest that edit sequences extracted from commits (C^{3}PO's original training data) may not represent the same edit sequences performed by developers in the IDE. Each edit sequence extracted from commits contained only edits spatially close in the file. Meanwhile, our dataset contains temporal sequences of edits applied by developers in the IDE, and thus, may contain non-local edits or other edits that are overwritten by the time the developer commits the code to the repository. While C^{3}PO's performance drastically increases when we use our training data, the overall performance is still low. We believe that the fact that C^{3}PO was not able to handle most of our data due to generative inserts, using only use 453 examples from our training data, is the main reason for that. Collecting more data may thus improve the accuracy of the model.

Additionally, extending the C^{3}PO model to support generative insertions where the new code is not presented in the input code snippet but exists in one of the previous edits would allow C^{3}PO to copy temporal information from previous edits similar to Overwatch. Investigating this direction, as well as exploring ways to combine the neural edit-completion model with symbolic techniques, are important directions for future work.

7.4.2 Comparison to Blue-Pencil. Blue-Pencil is a program synthesis based technique for producing repetitive edit suggestions in an IDE. As a developer is programming in an IDE, Blue-Pencil (a) observes the developer's changes, (b) synthesizes *suggestion programs* using repetitive edits that the developer is making as input-output examples, and (c) runs these suggestion programs to make suggestions at other similar locations in the code. Blue-Pencil is intended to be used in an *online* manner, i.e., only the developer's edits in the current session are used for learning; in contrast, Overwatch learns patterns offline over historical data. The suggestion programs learned by Blue-Pencil are meant to capture similar editing patterns as ESPs of the form et*, i.e., a single repetitive edit template, produced by Overwatch. To compare Overwatch with Blue-Pencil, we

[1] https://github.com/purug2000/TemporalC3PO
[2] https://github.com/dotnet/roslyn

run Blue-Pencil in an offline manner similar to Overwatch, learning suggestion programs from historical training data and testing them on a different test dataset.

Experimental Setup. We reuse the training dataset and test datasets consisting of 134,545 and 201,142 versions, respectively. One important hyper-parameter for Blue-Pencil is the *DAG size*, the size of the history over which Blue-Pencil looks for repetitive edits. Intuitively, with small DAG sizes, Blue-Pencil only searches for repetitive edit programs where the example edit instances are temporally close to each other in the training data, while they are allowed to be further apart for large DAG sizes. Given that Overwatch can handle patterns where the example instances come from anywhere in the training data, we leave the DAG size unbounded for our experiments.

Results. From the 135K versions of the training set, Blue-Pencil learned 587 unique suggestion programs. These 587 programs produced suggestions at 30,683 locations in the 201k versions of the test set of which only 177 were correct, leading to an extremely poor precision of 0.58%. In contrast, the ESPs of the form et* produced by Overwatch produced suggestions at 111 locations over the test set of which 86 were correct (precision = 77.48%). We also examined Blue-Pencil's per-program distribution of the number of suggestions and their precision. We found that 508 suggestion programs produced no suggestions at all on the test dataset. Of the 79 programs that did produce suggestions, 53 had a precision of 0, i.e., they did not produce a single correct suggestion. Overall, only 7 patterns had a precision of greater than 50%. Hence, most suggestion programs produced by Blue-Pencil either overfit the training data and do not produce any suggestions on the test data or produce too many incorrect, spurious suggestions on the test data. We found 2 main reasons for the poor performance of Blue-Pencil in this experiment.

Abstraction level of learned programs. Overwatch works at a higher level of abstraction than Blue-Pencil since the later completes a specific task that a user is doing, while the former learns patterns of common tasks that users perform in general. As such, ESPs and suggestion programs are defined at different levels of abstraction. For example, a Blue-Pencil suggestion program can express a specific renaming the user may be doing (say renaming method `m1` to `m2`), but cannot express the renaming task pattern in general. However, an ESP can capture the general renaming workflow that can produce suggestions whether a developer is renaming `m1` to `m2` or `m3` to `m4`: it uses the temporal context to capture the information that the developer is doing a particular rename in the substitution and can then produce suggestions for that particular renaming task. The specific programs that are learned by Blue-Pencil are unlikely to be useful beyond the current development session. For example, the program the renames `m1` to `m2` will not produce any suggestions if the test data does not contain code referring to the specific method `m1`. This is the reason that a vast majority (508) of the suggestion programs learned by Blue-Pencil in the experiment do not produce any suggestions at all.

Clustering and Filtering Techniques. Blue-Pencil selects programs by (i) filtering out programs that have low score according to a set of heuristics and (ii) selecting the remaining programs in the *least colorful path* of the DAG where these programs are stored. The least colorful path, in this context, is a path that uses the fewest programs to represent all the edits performed by the developers. By doing so, Blue-Pencil favors programs that are more general and thus can represent more edits over multiple, more specific, programs that cover only a subset of the edits. We noticed that this ranking and filtering approach led Blue-Pencil to synthesize many over-generalized suggestion programs. Most of the 53 of 79 Blue-Pencil programs that produce only incorrect suggestions fall into this category—they over generalize repetitive edits from different parts of the training data and Blue-Pencil fails to eliminate them during the filtering step. The Blue-Pencil approach works well for the online scenario (see Miltner et al. [2019]), where there are few examples and the technique needs to generalize fast but not for the offline scenario where the high number of edits leads to many incorrect clusters and over-generalization. Instead, Overwatch uses Agglomerative Hierarchical Clustering, which has the advantage of producing a hierarchy of clusters from the most specific to the more general ones, and ranks the ESPs based on their accuracy.

For example, our training dataset contained multiple sessions where developers delete multiple `throw` statements. Blue-Pencil clustered these edits with other delete statement edits, which led to an overgeneralized program that deletes any statement. Meanwhile, one of the clusters in the cluster hierarchy produced by Overwatch contained only the examples of deleting the `throw` statements. This cluster produced an ESP with a higher accuracy, given that the ESP only suggested to delete a `throw` statement if the developer had just deleted another one before that.

Overwatch outperforms both C^3PO and Blue-Pencil in our experiments. C^3PO does not support generative insertions thus rendering it incapable of handling 82% of our dataset, while Blue-Pencil fails to synthesize transformations at the right level of abstraction in an offline setting.

8 DISCUSSION

Our empirical study shows that Overwatch could benefit developers by predicting edits based on the temporal context. Overwatch uses temporal context to predict the next edit following the user's workflow using learned ESPs, rather than the developer having to invoke existing refactoring tools manually. This avoids to discoverability and late awareness problems discussed in Section 1. We implemented four ESPs to add temporal context for two existing refactorings in Visual Studio[3]—now, these refactorings are provided as suggestions in the developer's workflow when the developer starts performing these refactorings manually. Although these refactoring features existed before, we observed a 4x increase in the number of users using the tool to complete these tasks.

Overwatch synthesizes patterns from previous data and avoids the manual implementation of these patterns. It took us almost 800 lines of code and several iterations to get four patterns correct because of many entry points and corner cases. If we consider all the existing refactorings (>100 in Visual Studio) and all the different ways developers can perform each one of them, manually writing these patterns will not scale. Overwatch provides a way of avoiding this manual work.

8.1 Limitations

Our qualitative analysis revealed some limitations of Overwatch. First, we observed that some ESPs could be represented as a single more general ESP, which happened when two sketches could be generalized into a single sketch. A possible solution for this problem is to try to merge the top-ranked ESPs of each sketch in a second round of hierarchical clustering, filtering, and selection.

Additionally, we identified several transient and noisy ESPs. While only 13% of the ESPs were considered transient, and this number can be reduced by implementing a few heuristics based on these patterns, noisy ESPs represented 33.6% of the patterns. We observed that many of these noisy ESPs were related to patterns that were too specific, which would not produce false positives, but also would not likely trigger on other codebases. This problem can be alleviated with more data and a higher threshold for the support of each edge in the quotient graph (we used support = 2).

Finally, our ranking step (Section 6) will filter out ESPs with unbounded holes that cannot be filled by hole predicates because these ESPs will have zero precision. For example, our approach will filter out the ESP when user copies a property declaration and then tries to update the name of the copied property ($v_6 \rightarrow v_7 \rightarrow v_8$ in Figure 3) because we do not know what will be the new name of the property. To make these ESPs with unbound holes usable, a human-in-the-loop setup where the user could concretize these unbounded holes is a viable solution. Additionally, we rank

[3] https://devblogs.microsoft.com/visualstudio/just-in-time-refactoring-intellicode-suggestions/

our ESPs using their performance on the training dataset. However, we could rank them adaptively every time when making a prediction given the context of the code. We foresee that combining large language models with our ESPs will be a fruitful research direction because the language models can help fill in unbounded holes in ESPs and rank ESPs adaptively for each prediction.

8.2 Threats to validity

Construct Validity. We measured precision by checking if the edit predicted by Overwatch leads to the exact same code of a subsequent version. Some correct predictions, however, may lead to code that is similar but not exact the same. If the prediction adds a property to the beginning of the class but the developer ended up ending it somewhere in the middle, we will classify the prediction as a false positive, even though both edits are semantically equivalent. Therefore, our false positives may contain some true positives.

Internal Validity. The choice of some parameters used in our evaluation may impact the results. To reduce the bias on the choice of parameters, we performed a cross-validation to select the parameters ($threshold_1$ and $threshold_2$) that would impact the most on the precision of Overwatch.

External Validity. Overwatch learned ESPs from 682 code development sessions from 12 Visual Studio users (developers) that worked on 4 separate C# code bases. While our results suggest that the ESPs generalize to other datasets, such as our test dataset collected several months after the training dataset, these ESPs might not generalize to other IDEs or programming languages. Nevertheless, we identified several ESPs that have corresponding features in the IDE, and the proposed technique is independent of programming languages.

Verifiability. Our dataset is not publicly available due to a non-disclosure agreement with our participants.

9 RELATED WORK

Our work distinguishes itself from existing work by simultaneously **learning** (i) the edits from code development sessions in IDEs, and (ii) the temporal relation between the edits. Existing work on *edit patterns* is mostly focused on coarse-grained edits, whereas existing work on *edit sequence patterns* is limited in the scope of edits it considers.

Learning Edit Patterns from Commits Previous researchers have proposed a plethora of techniques that learn edits patterns from the coarse commit level changes. de Sousa et al. [2021] proposed a technique that infers code change pattern as rewrite rules (not specific fixes, or bugs) using anti-unification and a greedy algorithm for clustering. Similarly, Bader et al. [2019] proposed a technique (`Getafix`) to learn bug fix patterns using anti-unification. They presented a novel hierarchical, agglomerative clustering algorithm to cluster the examples. Getafix then applies an effective ranking technique that uses three metrics to produce a small and appropriate number of fixes for a given bug. `Getafix` inspires our design of the algorithms synthesizing edit sequence patterns, including anti-unification and the hierarchical clustering algorithm. However, Overwatch learns edit sequence patterns (not just an edit template) from fine-grained code development sessions instead of commit level data.

Recently, Ketkar et al. [2022] proposed `TCI-Infer` that learns the rewrite rules to perform type changes from the type changes identified by `RefactoringMiner`[Tsantalis et al. 2020] in the commit level history of Java projects. Similarly, `A3` [Lamothe et al. 2020] and `MEditor` [Xu et al. 2019] infer the adaptations required to perform library migration by analyzing the changed control/data flow across the commit. Kim et al. [2013] proposed a syntactic approach to automatically discover and represent systematic changes as logic rules with the goal to enhance developer's understanding about the program's evolution. Previously, Meng et al. [2011, 2013]; Ray et al. [2012] proposed techniques to perform systematic code changes by creating a context-aware edit script, finding

potential locations and transforming the code. Andersen and Lawall [2008] propose a technique that generates generic patches from a set of files and their updated versions and applies these to other files. Yin et al. [2019] propose a model that combines neural encoder with edit encoder, to express salient information of an edit and can be used to apply the edit. In contrast, Overwatch learns *sequences* of edit patterns (as opposed to a single edit) that developers often apply while performing their daily code development activities in an IDE. Blue-Pencil [Miltner et al. 2019] identifies repetitive changes, and automatically suggests similar repetitive edits. However, all these techniques either focus on a specific kind of edit or operate on coarse-grained VCS data which is *imprecise*, *incomplete* and makes it *impossible* to involve the temporal aspect [Negara et al. 2012].
Learning Edit Patterns from Sequence of Changes. Mesbah et al. [2019] propose `DeepDelta` to automatically suggest code fix for the common classes of build-time compilation failures. They encode the human-authored, in-progress changes into a domain-specific language and feed them to a neural machine translation network along with the compiler diagnostic. In contrast, Overwatch operates over finer IDE level sequences of code changes to learn the sequences of edits performed for a large variety of daily code development activities, not limited to compilation error fixes. Previously Negara et al. [2014] proposed a technique to detect *high-level* code change patterns from the *fine-grained* sequences of edits recorded in the IDE. In contrast, Overwatch learns sequences of *executable edit templates* that not only captures the *high-level* code change patterns but also the different workflows or sequences of edits that were applied to perform the change.
Detecting Sequences of Edit Patterns. Previous researchers have also tackled the temporal aspect of applying edits like Overwatch. [Foster et al. 2012; Ge et al. 2012] identified that *discoverability* and *late-awareness* led to *underuse* of refactoring tools, and proposed the techniques `Benefactor` and `WitchDoctor` to overcome it. The users of these technique have to manually encode the sequence of edits that are applied to perform a larger refactoring. The authors conducted interviews with software developers to manually recorded the sequences of edits they apply to perform a refactoring. These tools detect when the user is performing a refactoring, and suggest a completion. On the other hand Overwatch, automatically learns the sequence of edits that developers apply to perform any high level programming task (beyond refactorings) from code development sessions. Overwatch basically automates the entire pipeline proposed by `Benefactor` and `WitchDoctor`.

10 CONCLUSION

We introduce and tackle the problem of learning edit sequence patterns, with the aim of adding temporal context into IDE edit suggestion tools. ESPs capture fine-grained details in developers' editing behaviour, and can be used to address the two key challenges towards broader usage of IDE edit suggestion tools—discoverability and late awareness. Our experiments show that Overwatch can not only learn and automate editing patterns related to existing IDE tools, but also discover new patterns! Besides being useful to automatically make edit suggestions in the IDE, we foresee the ESPs being used by IDE toolsmiths to decide and prioritize what new features they should develop in a data-driven manner.

ESPs and temporal context can help develop tools that are more accurate, and in turn, allow for more aggressive presentation of suggestions with hand-raising interfaces like "grey text". We are currently exploring the design space for these interfaces to determine the best way to present different kinds of insert, delete, and update edit suggestions based on the IDE's confidence in those suggestions. With the advent of powerful pre-trained language models for source code has also opened up the possibility of extending Overwatch to cover more general patterns with unbounded holes. Overall, ESPs offer the possibility of developing a new generation of IDE tools and exploring a rich and exciting set of ideas from a range of research fields from HCI to AI to static analysis.

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, USA, 337–346. https://doi.org/10.1109/ASE.2008.44

Johannes Bader, Andrew Scott, Michael Pradel, and Satish Chandra. 2019. Getafix: Learning to Fix Bugs Automatically. *Proc. ACM Program. Lang.* 3, OOPSLA, Article 159 (Oct. 2019), 27 pages. https://doi.org/10.1145/3360585

R. Bloem, H. N. Gabow, and F. Somenzi. 2006. An algorithm for strongly connected component analysis in n log n symbolic steps. *Formal Methods in System Design* 28, 1 (2006), 37–56.

Shaked Brody, Uri Alon, and Eran Yahav. 2020. A structural model for contextual code changes. *Proc. ACM Program. Lang.* 4, OOPSLA (2020), 215:1–215:28. https://doi.org/10.1145/3428283

John L. Campbell, Charles Quincy, Jordan Osserman, and Ove K. Pedersen. 2013. Coding In-depth Semistructured Interviews. *Sociological Methods & Research* 42, 3 (2013), 294–320. https://doi.org/10.1177/0049124113500475

Nancy Chinchor. 1992. MUC-4 Evaluation Metrics. In *Proceedings of the 4th Conference on Message Understanding* (McLean, Virginia) *(MUC4 '92)*. Association for Computational Linguistics, USA, 22–29. https://doi.org/10.3115/1072064.1072067

William HE Day and Herbert Edelsbrunner. 1984. Efficient algorithms for agglomerative hierarchical clustering methods. *Journal of classification* 1, 1 (1984), 7–24.

Reudismam Rolim de Sousa, Gustavo Soares, Rohit Gheyi, Titus Barik, and Loris D'Antoni. 2021. Learning Quick Fixes from Code Repositories. In *SBES '21: 35th Brazilian Symposium on Software Engineering, Joinville, Santa Catarina, Brazil, 27 September 2021 - 1 October 2021*, Cristiano D. Vasconcellos, Karina Girardi Roggia, Vanessa Collere, and Paulo Bousfield (Eds.). ACM, 74–83. https://doi.org/10.1145/3474624.3474650

S. R. Foster, W. G. Griswold, and S. Lerner. 2012. WitchDoctor: IDE support for real-time auto-completion of refactorings. In *2012 34th International Conference on Software Engineering (ICSE)*. 222–232. https://doi.org/10.1109/ICSE.2012.6227191

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

Xi Ge, Quinton L. DuBose, and Emerson Murphy-Hill. 2012. Reconciling Manual and Automatic Refactoring. In *Proceedings of the 34th International Conference on Software Engineering* (Zurich, Switzerland) *(ICSE '12)*. IEEE Press, 211–221.

JetBrains. 2021. ReSharper. (2021). At https://www.jetbrains.com/resharper/.

Ameya Ketkar, Oleg Smirnov, Nikolaos Tsantalis, Danny Dig, and Timofey Bryksin. 2022. Inferring and Applying Type Changes. In *44th International Conference on Software Engineering (ICSE '22)* (Pittsburgh, United States) *(ICSE '22)*. ACM. https://doi.org/10.1145/3510003.3510115

M. Kim, D. Notkin, D. Grossman, and G. Wilson. 2013. Identifying and Summarizing Systematic Code Changes via Rule Inference. *IEEE Transactions on Software Engineering* 39, 1 (2013), 45–62. https://doi.org/10.1109/TSE.2012.16

Maxime Lamothe, Weiyi Shang, and Tse-Hsun Peter Chen. 2020. A3: Assisting Android API Migrations Using Code Examples. *IEEE Transactions on Software Engineering* (2020), 1–1. https://doi.org/10.1109/TSE.2020.2988396

Na Meng, Miryung Kim, and Kathryn S. McKinley. 2011. Sydit: creating and applying a program transformation from an example. In *SIGSOFT/FSE'11 19th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-19) and ESEC'11: 13th European Software Engineering Conference (ESEC-13), Szeged, Hungary, September 5-9, 2011*, Tibor Gyimóthy and Andreas Zeller (Eds.). ACM, 440–443. https://doi.org/10.1145/2025113.2025185

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

Ali Mesbah, Andrew Rice, Emily Johnston, Nick Glorioso, and Edward Aftandilian. 2019. DeepDelta: Learning to Repair Compilation Errors. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Tallinn, Estonia) *(ESEC/FSE 2019)*. Association for Computing Machinery, New York, NY, USA, 925–936. https://doi.org/10.1145/3338906.3340455

Microsoft. 2021. Visual Studio. (2021). At https://www.visualstudio.com.

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 (Oct. 2019), 29 pages. https://doi.org/10.1145/3360569

Emerson Murphy-Hill, Chris Parnin, and Andrew P. Black. 2009. How We Refactor, and How We Know It. In *Proceedings of the 31st International Conference on Software Engineering (ICSE '09)*. IEEE Computer Society, USA, 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* (Hyderabad, India) *(ICSE 2014)*. Association for Computing Machinery, New York, NY, USA, 803–813. https://doi.org/10.1145/2568225.2568317

Stas Negara, Mohsen Vakilian, Nicholas Chen, Ralph E. Johnson, and Danny Dig. 2012. Is It Dangerous to Use Version Control Histories to Study Source Code Evolution?. In *Proceedings of the 26th European Conference on Object-Oriented Programming*

(Beijing, China) *(ECOOP'12)*. Springer-Verlag, Berlin, Heidelberg, 79–103. https://doi.org/10.1007/978-3-642-31057-7_5

Gordon D. Plotkin. 1970. A Note on Inductive Generalization. *Machine Intelligence* 5 (1970), 153–163.

Baishakhi Ray, Christopher Wiley, and Miryung Kim. 2012. REPERTOIRE: A Cross-System Porting Analysis Tool for Forked Software Projects. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering* (Cary, North Carolina) *(FSE '12)*. Association for Computing Machinery, New York, NY, USA, Article 8, 4 pages. https://doi.org/10.1145/2393596.2393603

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 *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. IEEE, 404–415.

J Saldana. 2009. *The coding manual for qualitative researchers*. https://doi.org/10.1108/QROM-08-2016-1408

Nikolaos Tsantalis, Ameya Ketkar, and Danny Dig. 2020. RefactoringMiner 2.0. *IEEE Transactions on Software Engineering* (2020), 21 pages. https://doi.org/10.1109/TSE.2020.3007722

Mohsen Vakilian, Nicholas Chen, Stas Negara, Balaji Ambresh Rajkumar, Brian P. Bailey, and Ralph E. Johnson. 2012. Use, disuse, and misuse of automated refactorings. In *2012 34th International Conference on Software Engineering (ICSE)*. 233–243. https://doi.org/10.1109/ICSE.2012.6227190

C. J. van Rijsbergen. 1979. *Information Retrieval*. Butterworth. 133–134 pages. https://doi.org/10.1002/asi.4630300621 Available at http://www.dcs.gla.ac.uk/Keith/Preface.html.

Shengzhe Xu, Ziqi Dong, and Na Meng. 2019. Meditor: Inference and Application of API Migration Edits. In *Proceedings of the 27th International Conference on Program Comprehension* (Montreal, Quebec, Canada) *(ICPC '19)*. IEEE Press, Piscataway, NJ, USA, 335–346. https://doi.org/10.1109/ICPC.2019.00052

Pengcheng Yin, Graham Neubig, Miltiadis Allamanis, Marc Brockschmidt, and Alexander Gaunt. 2019. Learning to Represent Edits. In *ICLR 2019*. https://www.microsoft.com/en-us/research/publication/learning-to-represent-edits/ arXiv:1810.13337 [cs.LG].