Getafix

Learning to fix bugs automatically

Andrew Scott
Facebook
andrewscott@fb.edu

Johannes Bader
Facebook
jobader@fb.com

Satish Chandra
Facebook
satch@fb.com

Abstract

Static analyzers, including linters, can warn developers about programming errors, bugs, style errors, and generally suspicious code. Sometimes analyzer rules have obvious fixes but other times choosing an appropriate fix is more nuanced and can take some time for a developer to proceed. Offering fix suggestions for these trickier analyzers can therefore help save developer time, but in order for developers to accept the offered fix the patch must look natural. Additionally, in a production environment there is a limited amount of resources that can be spent validating fix candidates, so there is not time to evaluate a large number of fix candidates. In fact, at Facebook we can generally only afford to validate a single suggested fix. To solve these issues, we built Getafix, a tool which mines patterns from past fixes made by developers with enough context to accurately create natural-looking autofix suggestions. The patches we learn contain all the context needed to apply and rank them, so even though we only validate the top fix candidate we still are able to offer fix suggestions in a lot of cases. Additionally, our pattern mining approach can be directed at the whole code change history to find new lint warnings, avoiding the manual effort from developers to create new lint rules. Our pattern mining approach is more effective in discovering new and applicable patterns than previous work because it includes context about where and how the pattern should apply.

1 Introduction

Modern production codebases are extremely complex and are updated constantly. Static analyzers can help developers find potential issues (referred to as bugs for the rest of this paper) in their code, which is necessary to keep code quality high in these large codebases. Offering a fix along with a report of a bug can save developers time that they would otherwise need to spend thinking about how to fix the bug. Past human fixes of the same bug offer good insight into how future instances of the bug should be fixed, so we built a tool that mines patterns from those previous changes and uses them to identify the most likely remediation for new bugs. Our goal is to let computers take care of the routine work, albeit under the watchful eye of a human, who must decide when a bug requires a complex, nonroutine remediation.

1.1 Getafix

This tool, called Getafix, has been deployed to *production* at Facebook, where it now contributes to the stability of apps that billions of people use. In our industrial setting it costs significant amounts of time and money to validate changes. At the same time, suggestions should be available as quickly as possible and satisfy a human reviewer to the extend that they are willing to accept them, i.e. suggestions should look as if they were human-written or our service does not save developers' time. We compile these expectations into the following key challenges:

- Getafix has *one shot* at producing a fix suggestion that passes validation
- A Getafix suggestion should look *natural* to a human reviewer
- Getafix should produce its suggestion as fast as possible

To address these challenges, we believe that *patch prioritization*Âăis important to reduce the number of potential validations, as shown by Long and Rinard [9]. Since Getafix only gets one shot at passing validation and pleasing the human reviewer, it must have high confidence in its suggestion.

Furthermore we believe that the *naturalness* of patches is key to user acceptance, which means that we need to learn a rich library of fix patterns which not only capture the patch to perform, but also enough context to judge which pattern would be the most natural one. We build upon previous research on history-based repair like [7], which has a good chance of producing natural patches.

Narrowing the search space. Note the stark contrast of our setting to *generate-and-validate* systems for automated bug fixing which rely on validating a potentially large set of fix candidates [3][5][7]. Furthermore, this work usually assumes no prior knowledge about the *root cause* of the failures it attempts to fix, for instance when trying to make failing test cases succeed. This problem spawned great research on fault-localizers.

Getafix on the other hand addresses problems at a known location, which we assume have a root cause in the same file and most likely in close proximity of the blamed line. However, given that there might be many fix patterns and the neighborhood of a reported bug might still produce a large number of fix candidates.

Due to the potentially long time to find a valid fix candidate, a limitation that also motivated Hua et al. [4], *generate-and-validate* approaches are not applicable to our production setting without further modifications or improvements. A subset of this existing work demarcates between fix templates (e.g. "enrich a conditional") and fix ingredients (e.g. the exact construct to add to a conditional). Guessing fix ingredients involves heuristics like combinatorial search, which can result in a plethora of fix candidates that we cannot afford to validate. Getafix works when the fix ingredient can be *determined* rather than guessed, and employs several strategies to do so. Our work hence no longer speaks about fix ingredients in the first place, we talk about *fix patterns* which are applicable to new code out of the box.

Pipeline. We offer a deep dive into how Getafix fixes bugs (using the term broadly to refer to any code issues, not just those that will cause an app to crash).

Getafix creates a *database of fix patterns* by learning from past human code changes. As training data can serve any collection of past human code changes tied to a certain signal. This signal could be disappearance of code quality issues flagged by another tool (static analysis warnings, type errors, lint messages) or simply the fact that a change was tied to a human code review (suggested change). We improved the learning process by applying a new method of hierarchical clustering to many thousands of past code changes that human engineers made. Getafix learns from both the code change itself and additional context around it (surrounding AST nodes, bug report). This method allows it to detect the underlying patterns in bugs and the corresponding fixes that previous auto-fix tools could not and that result in more natural fixes.

Presented with a new bug report and buggy code, Getafix will use the learnt fix patterns to *predict the most likely patch* which is subsequently validated. Due to limited validation budget, Getafix has one shot at producing the right fix, so it must have high confidence in it. For this purpose, Getafix internally ranks candidate patches by estimating the probability of being the most natural fix. We present and evaluate a good estimation of this probability, leveraging the additional context mined with fix patterns. This work is not concerned with fault localization since the bug reports Getafix reacts to already blame a specific line of code (as one can imagine from linters or static analysis tools).

Validation of the predicted patch is typically realized using the same tool that provided signal for the training data: static analysis, type checker, linter. Getafix is agnostic to the training/validation signal used, so this work does not focus on validation per se. We assume validation is a black box component that is expensive to invoke.

Deployment. At Facebook, Getafix currently suggests fixes for bugs found by Infer[1], our static analysis tool that identifies issues such as null pointer exceptions in Android and Java code. It also suggests fixes âĂŤ via SapFix[2] âĂŤ for bugs detected by Sapienz[3], our intelligent automated testing system for apps. It is also being used to resolve code quality concerns en masse that are found when revisiting existing code with newer versions of Infer.

1.2 Contributions

To address the challenges of our industrial setting, Getafix makes the following contributions:

- Getafix patches contain all the information needed to apply them, and one-shot verification suffices as opposed to a search.
- Getafix patch application uses a sophisticated ranking method, that strives to find a natural patch.
- Getafix patch mining is effective in discovery of natural patches, owing to a hierarchical clustering approach.

The experiments in section 5 will support these claims.

How Getafix differs from simpler industrial auto-fix tools. In current industrial practice, auto-fixes have been used primarily for basic issues, whereas code remediation is straightforward. For example, an analyzer might warn about a âĂIJdead exception,âĂİ in which the developer probably forgot to add a `throw` before `new Exception(...)`. An auto-fix to make that change is straightforward and can be defined by the author of the lint rule, without knowing the specific context in which it is applied.

Getafix offers significantly more general capability, remediating issues in cases where the fix is context-dependent. This context-dependence addresses the challenges of our setting by not only producing more specialized and likely valid, but also more natural fixes. The following example fix (anonymized) was suggested by Getafix and accepted by an engineer, and demonstrates this sensitivity.

Example 1.1 (Context-dependent, natural fix)**.**

```
boolean process() {
  if (this.lazyProvider == null || manager.get() == null) {
    return false;
  }
  Provider p = this.lazyProvider.get();
  // ...
}
```

Static analysis detects that the framed expression can be `null`, so Getafix suggests inserting the highlighted snippet. Note that this fix depends not only on the blamed expression

[1]https://code.fb.com/developer-tools/open-sourcing-facebook-infer-identify-bugs-before-you-ship/
[2]https://code.fb.com/developer-tools/finding-and-fixing-software-bugs-automatically-with-sapfix-and-sapienz/
[3]https://code.fb.com/developer-tools/sapienz-intelligent-automated-software-testing-at-scale/

`this.lazyProvider` but also on surrounding existing structures. The fix pattern that Getafix selected here is aware of the existing if statement located above the bug and within the same block, as well as the fact that the statement guards a return statement. Hence, disjunctively extending the condition with a null-check prevents a potential `NullPointerException` with an early return. This pattern has a high chance to appeal to a human reviewer given that the alternative control flow *already existed*, furthermore an engineer already made the decision about the exact return value to use. Without this if statement, Getafix may have recommended:

```
boolean process() {
  if (this.lazyProvider == null) {
    return false;
  }
  Provider p = this.lazyProvider.get();
  // ...
}
```

Note that the this fix also depends on the return type of the method.

Unlike simple lint remediations, fixes of these kinds cannot be baked into static analysis tools themselves. Section 5 has additional examples of fixes that Getafix offered and that were accepted.

1.3 Architecture

The Getafix toolchain consists of three main components. In the following we will describe their functionality and challenges. Section 2 outlines the *tree differencer* used to identify changes at the AST level. The generated *concrete edits* are pairs of sub-ASTs of the original *before* and *after* ASTs, representing a specific mutation that could be replayed on different code by replacing instances of the edit's *before* AST with its *after* AST. In section 3 we explain a new way of *mining fix patterns* based on hierarchical agglomerative clustering. Fix patterns are represented the same way as concrete edits, only that "holes" (pattern variables) may occur that will not only match one specific AST structure (similar to representation used by Long et al. [8]). Section 4 focusses on the challenges around *creating and suggesting patches*, which involves ranking among countless candidate fixes.

In section 5 we present our experimental results, discuss related work in section 6 and future work and open questions in section 7.

2 Tree differencer

An abstract-syntax-tree-based differencer derived from the GumTree algorithm [2] is first used to identify concrete edits made between a pair of source files, such as successive revisions of the same file. For example, it will detect granular edits: wrapping a statement with an if, adding a `@Nullable` annotation or an `import`, and prepending a conditional early `return` to an existing method body, among others. In Figure 1, the insertion of a conditional early return if `task` is `null`, the rename of `public` to `private`, and the move of a method were detected as concrete edits. Whereas a line-based diffing tool would mark either method as fully removed and inserted, the tree differencer detects the move and can hence also detect the insertion within the moved method as a concrete edit.

```
public class Worker {
  public void doWork() {
    task.makeProgress();
  }
  public long getRuntime() {
    return now - start;
  }
}
```

↦

```
public class Worker {
  private long getRuntime() {
    return now - start;
  }
  public void doWork() {
    if (task == null)
      return null;
    task.makeProgress();
  }
}
```

Figure 1. Concrete edits detected by the tree differencer.

A challenge in the tree differencer is to efficiently and precisely align parts of the âĂIJbeforeâĂİ and âĂIJafterâĂİ trees, so the right concrete edits and mappings from *before* to *after* nodes get discovered. An unmapped node in the *before* or *after* tree is considered a *deletion* or *insertion*, respectively. A mapped pair of nodes whose parents are not also mapped to each other is considered a *move*. A mapped pair of nodes with different values (see change from `public` to `private` in Figure 1) is considered an *update* and may be part of a move at the same time. Otherwise (parents are mapped to each other and no change in value), the pair of mapped nodes is considered *unmodified*. An entire subtree is considered *unmodified* if all its nodes are unmodified. In Figure 1, the subtree representing `task.doWork();` is unmodified, while the subtree representing the surrounding block is not (due to the insertion). Unchanged subtrees have the useful property that dropping them from an edit has no effect on the modifications represented/performed by that edit. However, they carry useful information for applying an edit in previously unseen code, which will become important in later sections.

2.1 Definitions

For our purposes, an abstract syntax tree (AST) node has:

- a *label*, e.g. "BinEx" (for binary expression), "Literal" or "Name"
- a *value* (may be empty), e.g. `+`, `42` or `foo`, respectively
- *child nodes*, each having a *location* that describes their relationship to the parent node, e.g. "left" and "right" to address the left/right subexpression of a binary expression.

Definition 2.1 (Set of trees, edits, mappings)**.** We define the sets of ASTs, edits and mappings as

$$\begin{aligned}\mathsf{Tree} &= \mathsf{Label} \times \mathsf{Value} \times \bigcup_{k \in \mathbb{N}} (\mathsf{Location} \times \mathsf{Tree})^k \\ \mathsf{Edit} &= \mathsf{Tree} \times \mathsf{Tree} \times \mathcal{P}^{\mathsf{Mapping}} \\ \mathsf{Mapping} &= \mathsf{TreeRef} \times \mathsf{TreeRef} \times \{\mathtt{tt}, \mathtt{ff}\}\end{aligned}$$

where Label, Location and Value are sets of strings. While these three sets would likely be smaller or finite for specific programming languages, we make no such assumptions. Edits are triples containing the *before* and *after* ASTs, followed by the set of mappings. A mapping is a triple referencing a node of the *before* and a node of the *after* AST, as well as a flag indicating whether the pair of nodes is part of a modification (`tt` denotes participation in a modification, `ff` denotes that the mapped subtree is unmodified). TreeRef is the set of references uniquely identifying a node within an AST.[4]

Definition 2.2 (Edit well-formedness)**.** For an edit to be well-formed, each of its mappings need to meet the following requirements:

- The two node references resolve to a node of the *before* and *after* tree, respectively.
- If the flag indicates that the mapped subtrees are unmodified (`ff`):
  - then the two referenced nodes must be roots of identical subtrees
  - all descendants of those subtrees are also mapped and marked as unmodified
  - the parents of the referenced nodes must also be mapped (otherwise the node was *moved* and hence marked as modified)
- If sibling nodes change relative position (e.g. arguments of a function call swapped) then either of them must be classified as a *move* by the tree differencer and thus be marked as modified.

2.2 Notation

For brevity and readability, we represent ASTs using term notation rather than tuples. Parsing `x = y + 2` results in an AST like $\mathbf{Assign}(\mathbf{x : Name}, \mathbf{+ : BinEx}(\mathbf{y : Name}, \mathbf{2 : Literal}))$. Child nodes are listed like parameters of a function call, having the node's label as function name. Parentheses are omitted if no child nodes exist. Values (if not empty) are prepended in front of labels using syntax borrowed from type judgments. We omit stating locations of child nodes and assume they are obvious from the order in which child nodes are stated (e.g. left subexpression before right subexpression of a binary expression).

[4] The exact representation does not matter for our purposes. One could use vectors of indices representing the children to select while walking down the tree. In a language like Java one could simply use references to the very nodes one wants to address.

We write edits in source code as `x = y + 2` $\rightarrowtail$ `x = 3 + y`. A notation for AST edits as defined above must be able to express mappings between *before* and *after* tree, as well as mark subtrees as unmodified. Two concrete edit emitted for the above change are $\mathbf{+ : BinEx}_0(\mathbf{y : Name}_1, \mathbf{2 : Literal}) \rightarrowtail \mathbf{+ : BinEx}_0(\mathbf{3 : Literal}, \mathbf{y : Name}_1)$ and $\mathbf{Assign}_0(\mathsf{x : Name}_1, \mathbf{+ : BinEx}_2(\mathbf{y : Name}_3, \mathbf{2 : Literal})) \rightarrowtail \mathbf{Assign}_0(\mathsf{x : Name}_1, \mathbf{+ : BinEx}_2(\mathbf{3 : Literal}, \mathbf{y : Name}_3))$. We write unmodified subtrees non-bold and give nodes connected by a mapping matching indices that have to be unique only within the edit.

3 Learning patterns

Getafix performs pattern mining by using a new hierarchical clustering technique, along with anti-unification [6], an existing method of generalizing among different symbolic expressions. Given a set of concrete edits, anti-unification is iteratively applied to the (semantically) most similar pair of edits. Each iteration yields the generalization of the two anti-unified edits, which is added back in place of those two edits. These generalizations can be abstract edit patterns, containing âĂIJholesâĂİ (pattern variables) where program transformations differ.

As we merge concrete edits, we maintain the hierarchy showing the order in which they merge in a dendrogram structure. At the leaves of the dendrogram are the original concrete edits and at each higher level in the hierarchy is the result of anti-unifying the children from the previous level. As a result, the dendrogram structure we create contains edit patterns with varying levels of generality, from concrete edits at the leaves to abstract edits at the roots.

While the idea of anti-unification-based pattern mining has already been explored by Rolim et al. [16], several enhancements around learning the context in which edits are made were necessary to mine patterns that can be used to generate and rank a reasonably small number of fixes for a new bug.

3.1 Anti-unification

Anti-unification [6] is a process which takes two or more trees as input and produces a generalization that can describe all of the trees. The input trees are composed of constant symbols and the output generalization is composed of both constant symbols and variable symbols (or âĂIJholesâĂİ). The generalization is constructed such that each hole has a set of substitutions that can be used to recover the original input trees.

There always exists a generalization to describe multiple trees: a generalization that is just a single hole at the root. However, that generalization is generally not very interesting and therefore anti-unification seeks to find the least general generalization that can describe the input trees, meaning the generalization retains as much information as possible about the input trees.

3.1.1 Trees

To represent ASTs with holes we extend Tree with the set of holes Hole, resulting in the set of *tree patterns* Tree^+:

$$\begin{aligned}\mathsf{Tree}^+ &= \mathsf{Label} \times \mathsf{Value} \times \bigcup_{k\in\mathbb{N}} (\mathsf{Location} \times \mathsf{Tree}^+)^k \\ &\quad \cup \mathsf{Hole} \\ \mathsf{Hole} &= (\mathsf{Label} \cup \{?\}) \times \mathbb{N}\end{aligned}$$

Holes may have a label α (we write $\boldsymbol{h}^k : \boldsymbol{\alpha}$) in which case they match an arbitrary subtree that has a root with label α. If the label is omitted (we write $\boldsymbol{h}^k : \mathbf{?}$) the hole matches any subtree. Holes are indexed for identification purposes, allowing tree patterns to express whether two holes must match identical subtrees. Note that holes do not have children (which could further restrict what subtrees can be matched) in our setting.

Example 3.1 (Matching a tree pattern against a tree)**.** We define $t_1, t_2, t_3 \in \mathsf{Tree}$ and $p_1, p_2, p_3, p_4 \in \mathsf{Tree}^+$ as

$$\begin{aligned}t_1 &= \mathbf{+ : BinEx}(\mathbf{x : Name}, \mathbf{42 : Literal}) \\ t_2 &= \mathbf{+ : BinEx}(\mathbf{x : Name}, \mathbf{y : Name}) \\ t_3 &= \mathbf{+ : BinEx}(\mathbf{x : Name}, \mathbf{x : Name}) \\ p_1 &= \mathbf{+ : BinEx}\big(\boldsymbol{h}^0 : \mathbf{Name}, \boldsymbol{h}^1 : \mathbf{Literal}\big) \\ p_2 &= \boldsymbol{h}^0 : \mathbf{BinEx} \\ p_3 &= \mathbf{+ : BinEx}\big(\boldsymbol{h}^0 : \mathbf{Name}, \boldsymbol{h}^1 : ?\big) \\ p_4 &= \mathbf{+ : BinEx}\big(\boldsymbol{h}^0 : ?, \boldsymbol{h}^0 : ?\big)\end{aligned}$$

Then p_1 matches only t_1 since t_2 and t_3 do not match the label of hole $\boldsymbol{h}^1$. $\boldsymbol{p_2}$ and $\boldsymbol{p_3}$ match all three ASTs. $\boldsymbol{p_4}$ matches only $\boldsymbol{t_3}$ since it requires both operands of the binary expression to be identical. We will formalize what it means for a pattern to match a tree later.

Definition 3.2 (Anti-unification of tree patterns)**.** For conciseness, we assume the function performs memoization which causes it to reuse holes where possible, without explicitly tracking them.

$$\mathsf{antiUnify} : \mathsf{Tree}^+ \times \mathsf{Tree}^+ \longrightarrow \mathsf{Tree}^+$$

where

$$\mathsf{antiUnify}(\ \boldsymbol{v} : \boldsymbol{\alpha}(\boldsymbol{a_1}, \dots, \boldsymbol{a_n})\ ,\ \boldsymbol{w} : \boldsymbol{\beta}(\boldsymbol{b_1}, \dots, \boldsymbol{b_m})\ ) =$$

$$\begin{cases} \boldsymbol{v} : \boldsymbol{\alpha}(\boldsymbol{c_1}, \dots, \boldsymbol{c_n}) & \text{if } \alpha = \beta \wedge v = w \wedge n = m \\ \qquad \text{where } c_i = \mathsf{antiUnify}(a_i, b_i) \text{ f.a. } i \in \{1, \dots, n\} & \\ \boldsymbol{h}^k : \boldsymbol{\alpha} & \text{if } \alpha = \beta \wedge (v \neq w \vee n \neq m) \\ \boldsymbol{h}^k : \mathbf{?} & \text{otherwise} \end{cases}$$

$$\begin{aligned}&\text{for } n, m \in \mathbb{N},\ \ a_1, \dots, a_n, b_1, \dots, b_m \in \mathsf{Tree}^+ \\ &\quad \alpha, \beta \in \mathsf{Label},\ \ v, w \in \mathsf{Value} \\ &\quad \text{fresh hole index } k \in \mathbb{N}\end{aligned}$$

$$\begin{aligned}&\mathsf{antiUnify}(\ \boldsymbol{v} : \boldsymbol{\alpha}(\boldsymbol{a_1}, \dots, \boldsymbol{a_n})\ ,\ \boldsymbol{h}^t : \boldsymbol{\beta}\ ) = \\ &\mathsf{antiUnify}(\ \boldsymbol{h}^s : \boldsymbol{\alpha}\ ,\ \boldsymbol{w} : \boldsymbol{\beta}(\boldsymbol{b_1}, \dots, \boldsymbol{b_m})\ ) = \\ &\mathsf{antiUnify}(\ \boldsymbol{h}^s : \boldsymbol{\alpha}\ ,\ \boldsymbol{h}^t : \boldsymbol{\beta}\ ) =\end{aligned}$$

$$\begin{cases} \boldsymbol{h}^k : \boldsymbol{\alpha} & \text{if } \alpha = \beta \\ \boldsymbol{h}^k : \mathbf{?} & \text{otherwise} \end{cases}$$

$$\begin{aligned}&\text{for } n, m, s, t \in \mathbb{N},\ \ a_1, \dots, a_n, b_1, \dots, b_m \in \mathsf{Tree}^+ \\ &\quad \alpha, \beta \in \mathsf{Label},\ \ v, w \in \mathsf{Value} \\ &\quad \text{fresh hole index } k \in \mathbb{N}\end{aligned}$$

Anti-unification represents a join/least upper bound operation on Tree^+. The resulting semilattice $\langle \mathsf{Tree}^+, \mathsf{antiUnify}\rangle$ has Tree as minimal elements and $\boldsymbol{h}^k : \mathbf{?}$ (regardless of k) as its greatest element. Unification as known from logic programming and type systems would be the corresponding (partial) greatest lower bound operation.

Definition 3.3 (Tree pattern equality)**.** Two tree patterns $p_1, p_2 \in \mathsf{Tree}^+$ are considered equal if they are structurally identical, modulo bijective substitution of hole indices. E.g., $\mathbf{+ : BE}\big(\boldsymbol{h}^0 : \mathbf{Name}, \boldsymbol{h}^1 : ?\big) = \mathbf{+ : BE}\big(\boldsymbol{h}^5 : \mathbf{Name}, \boldsymbol{h}^3 : ?\big)$ and $\mathbf{+ : BE}\big(\boldsymbol{h}^0 : \mathbf{Name}, \boldsymbol{h}^1 : ?\big) = \mathbf{+ : BE}\big(\boldsymbol{h}^1 : \mathbf{Name}, \boldsymbol{h}^0 : ?\big)$ but $\mathbf{+ : BE}\big(\boldsymbol{h}^0 : \mathbf{Name}, \boldsymbol{h}^1 : ?\big) \neq \mathbf{+ : BE}\big(\boldsymbol{h}^2 : \mathbf{Name}, \boldsymbol{h}^2 : ?\big)$. As a consequence, all single-hole patterns $\boldsymbol{h}^k : \mathbf{?}$ are identical.

As mentioned earlier, we also track substitutions per hole during anti-unification, which not only allows reversing anti-unification but is also the mechanism realising the memoization we omitted above. We can now formalize what it means for an tree pattern to match a tree, using an auxiliary relation.

Definition 3.4 (Tree pattern precision)**.** $\langle \mathsf{Tree}^+, \mathsf{antiUnify}\rangle$ induces a partial order $\sqsubseteq$ on Tree^+ (read "more precise/specific than") by defining $a \sqsubseteq b \iff \mathsf{antiUnify(a, b)} = b$.

Definition 3.5 (Matching and substitutions)**.** Let $p \in \mathsf{Tree}^+$ and $t \in \mathsf{Tree}$. Then p matches t exactly if $t \sqsubseteq p$. In this case, there exists a function *subst* : Hole $\longrightarrow$ Tree capturing the substitutions one has to perform to turn p back into t.

Specifically, replacing every hole h in p with $subst(h)$ yields t.

3.1.2 Edits

We define the set of edit patterns Edit^+ analogous to Edit (see definition 2.1), but using Tree^+ instead of Tree. When anti-unifying two concrete edits, we first anti-unify the *before* trees of the concrete edits and then the *after* trees. We use a single set of substitutions between the *before* and *after* trees, meaning we can tell when a hole in the *before* tree corresponds to the same AST node in the *after* tree. In addition, we treat AST nodes which are modified between the *before* and *after* as part of a concrete edit differently from AST nodes which are unmodified. Specifically, when performing the recursive anti-unification algorithm on two trees, we pair up their modified children and then, if possible, also pair up unmodified children that exist in both trees. As we will discuss later, including these unmodified nodes in the patterns we learn allows us to learn the context in which certain edits are performed, which helps to narrow down the number of places a pattern can apply.

The first step of anti-unification is defined as follows.

$$\begin{aligned}
&\mathsf{antiUnify}(\ before_1 \rightarrowtail after_1\ ,\ before_2 \rightarrowtail after_2\ ) = \\
&\quad \mathsf{antiUnify}(\mathsf{stripUnch}(before_1), \mathsf{stripUnch}(before_2)) \rightarrowtail \\
&\quad \mathsf{antiUnify}(\mathsf{stripUnch}(after_1), \mathsf{stripUnch}(after_2)) \\
&\qquad \text{for } before_1 \rightarrowtail after_1, before_2 \rightarrowtail after_2 \in \mathsf{Edit}^+
\end{aligned}$$

where

$$\begin{aligned}
&\mathsf{stripUnch}(\boldsymbol{v} : \boldsymbol{\alpha}(\overline{\boldsymbol{a}_{1..i-1}}\ , a_i\ , \overline{\boldsymbol{a}_{i+1..n}})) = \\
&\quad \mathsf{stripUnch}(\boldsymbol{v} : \boldsymbol{\alpha}(\overline{\boldsymbol{a}_{1..i-1}}, \overline{\boldsymbol{a}_{i+1..n}})) \\
&\qquad \text{(drop unmodified children)} \\
&\mathsf{stripUnch}(\boldsymbol{v} : \boldsymbol{\alpha}(\boldsymbol{a}_1, \dots, \boldsymbol{a}_n)) = \boldsymbol{v} : \boldsymbol{\alpha}\left(\boldsymbol{a}'_1, \dots, \boldsymbol{a}'_n\right) \\
&\qquad\qquad \text{where } \boldsymbol{a}'_i = \mathsf{stripUnch}(\boldsymbol{a}_i)\ \text{ f.a. } \ i \in \{1, \dots, n\} \\
&\qquad \text{(if no more children are unmodified, recurse)} \\
&\qquad \text{for } n, i \in \mathbb{N}, \quad a_1, \dots, a_n \in \mathsf{Tree}^+ \\
&\qquad\quad \alpha \in \mathsf{Label}, \quad v \in \mathsf{Value}
\end{aligned}$$

Results of anti-unification between trees are memoized *across* the two function calls (but not across multiple invocations of anti-unification between edits). Mappings are established between any pair of nodes that are the result of anti-unifying mapped nodes. In other words, mappings are maintained where both edits agree about the existence of a mapping, otherwise dropped. In the second step we attempt to anti-unify and carry across as many unmodified nodes as possible. Again, existing holes (from the first step) are reused where possible.

We define edit pattern precision analogous to definition 3.4:

Definition 3.6 (Edit pattern precision)**.** $\langle \mathsf{Edit}^+, \mathsf{antiUnify}\rangle$ induces a partial order $\sqsubseteq$ on Edit^+ (read "more precise/specific than") by defining $a \sqsubseteq b \iff \mathsf{antiUnify(a, b)} = b$.

Example 3.7 (Anti-unification of two blocks of statements)**.** Assume the training data contains two source edits: `{ f(); g(); x = 1; }` $\rightarrowtail$ `{ f(); x = 1; if (c) g(); }` and `{ return; y = 2; }` $\rightarrowtail$ `{ y = 2; if (c) onResult(); }`. Corresponding concrete edits will instantiate

$$before_1 = \mathbf{B}_0(\mathsf{f} : \mathsf{CS}_1, \mathbf{g : CS}_2, \mathsf{AS}_3(\mathsf{x} : \mathsf{N}_4, 1 : \mathsf{Num}_5))$$
$$after_1 = \mathbf{B}_0(\mathsf{f} : \mathsf{CS}_1, \mathsf{AS}_3(\mathsf{x} : \mathsf{N}_4, 1 : \mathsf{Num}_5), \mathbf{IS}(\mathbf{c : N}, \mathbf{g : CS}_2))$$
$$before_2 = \mathbf{B}_6(\mathbf{RS}, \mathsf{AS}_7(\mathsf{y} : \mathsf{N}_8, 2 : \mathsf{Num}_9))$$
$$after_2 = \mathbf{B}_6(\mathsf{AS}_7(\mathsf{y} : \mathsf{N}_8, 2 : \mathsf{Num}_9), \mathbf{IS}(\mathbf{c : N}, \mathbf{onResult : CS}))$$

(Labels are abbreviations for block, call statement, assignment statement, name, number literal, if statement and return statement, chronologically.)

In the first step we drop the unmodified subtrees and anti-unify the result:

$$\begin{aligned}
&\mathsf{stripUnch}(before_1) = \mathbf{B}_0(\mathbf{g : CS}_1) \\
&\mathsf{stripUnch}(after_1) = \mathbf{B}_0(\mathbf{IS}(\mathbf{c : N}, \mathbf{g : CS}_1)) \\
&\mathsf{stripUnch}(before_2) = \mathbf{B}_6(\mathbf{RS}) \\
&\mathsf{stripUnch}(after_2) = \mathbf{B}_6(\mathbf{IS}(\mathbf{c : N}, \mathbf{CS})) \\
&\mathsf{antiUnify}(\ before_1 \rightarrowtail after_1\ ,\ before_2 \rightarrowtail after_2\ ) = \\
&\quad \mathbf{B}\big(\boldsymbol{h}^0 : ?\big) \rightarrowtail \mathbf{B}\big(\mathbf{IS}\big(\mathbf{c : N}, \boldsymbol{h}^1 : \mathbf{CS}\big)\big)
\end{aligned}$$

In this case, mappings 0 and 6 agree with each other, resulting in $\mathbf{B}_{10}\big(\boldsymbol{h}^0 : ?\big) \rightarrowtail \mathbf{B}_{10}\big(\mathbf{IS}\big(\mathbf{c : N}, \boldsymbol{h}^1 : \mathbf{CS}\big)\big)$.

In the second step we try anti-unifying and adding back unmodified nodes. In this case this succeeds for the assignment statements, ultimately resulting in
$\mathbf{B}_{10}\big(\boldsymbol{h}^0 : ?, \mathsf{AS}_{11}\big(h^2 : \mathsf{N}, h^3 : \mathsf{Num}\big)\big) \rightarrowtail$
$\mathbf{B}_{10}\big(\mathsf{AS}_{11}\big(h^2 : \mathsf{N}, h^3 : \mathsf{Num}\big), \mathbf{IS}\big(\mathbf{c : N}, \boldsymbol{h}^1 : \mathbf{CS}\big)\big)$.

If the details of AST representation are not relevant, we will from now on also represent edits patterns like the above using pseudo-code, e.g. `h0; h2 = h3;` $\rightarrowtail$ `h2 = h3; if (c) h1();`. This notation is often more natural to deal with but omits a number of details. We will mention details like node labels or unmodified nodes where necessary.

3.2 Hierarchical clustering

Hierarchical clustering iteratively applies anti-unification to the most similar pair of edits, which is the pair where anti-unification results in the least loss of information. For this purpose we first formalize our metric used to determine the pair of edits to merge and give an example showing the superiority of using hierarchical clustering with this metric rather than using greedy clustering. We then discuss efficient implementation options of hierarchical clustering that do not require finding the optimal merge out of all possible pairs on each iteration. Finally, we summarize desirable properties of our technique.

Merge order. The order in which we merge edits during hierarchical clustering is derived from the anti-unification process described in section 3.1. Given some set I and edits $a_i, b_i \in \mathsf{Edit}^+$ for all $i \in I$, which represent candidate merges, let $c_i = \mathsf{antiUnify}(a_i, b_i)$. Trying to preserve as much information as possible, we prefer merges that retain the precision of edit patterns when possible. For this purpose we impose a partial order on I by applying $\sqsubseteq$ (see definition 3.6) to the merge results: For any $i, j \in I$ we define $i \sqsubseteq j \Leftrightarrow c_i \sqsubseteq c_j$. The minimal elements of I represent merges preserving the most information, so one of them should be performed next.

Figure 2 visualizes an example dendrogram created from four concrete edits. Note how most similar edits got merged iteratively and patterns get more general on higher levels of the dendrogram (further left).

Implementation. Since $\sqsubseteq$ is expensive to compute, we approximate it heuristically using the following steps, each acting as a tiebreaker for the previous step. We prefer generalizations which

1. are closed over open ones (see Figure 3a for an example)
2. preserve more mappings between modified nodes
3. have less holes, and holes generalizing away smaller subtrees
4. preserve more mappings between unmodified nodes with labels (anti-unification of two nodes with different labels will produce a result with no label)
5. preserve more error context (explained in section 3.3)
6. preserve more mappings between unmodified nodes

Note that after performing one merge, the new set of edit patterns and hence also the new set of merge candidates is largely unchanged. We therefore use the nearest-neighbor-chain algorithm [1], which reduces the number of times the same pair of potential merges is compared.

In order use the hierarchical clustering approach (which is an n^2 algorithm at its heart) on large datasets, we cannot try to construct a single hierarchical structure containing every single edit at once. Instead, we start by partitioning the edits by the labels of their modified nodes in the *before* and *after* trees. Edits for which the labels differ get least priority in the merge order explained previously, so a lot of time can be saved by putting them into separate partitions and constructing a dendrogram for each partition individually. Finally, the roots of these dendrograms are further combined to form a single dendrogram.

Advantages of this approach. A greedy clustering algorithm, as suggested in past literature on auto-fix tools [16], is unlikely to learn patterns with the level of context that we discover. This is because a greedy clustering algorithm maintains a single representation of each cluster, which will not include the extra context if it is not present in all of the edits in the training data. For example, in Figure 2, the edit at the top of the hierarchy simply says to insert a check for null before something (the "something" represented by `h0`). This is the only pattern that would be learned if we only had a single representation of this cluster, but since we have the hierarchy we are able to observe the pattern one level down where the early return is inserted before a method call which uses the variable that could be null (`h0.h1()`). This extra context increases the probability that the first suggestion we make will fix the bug in question.

Candidate patches are created from all patterns in the dendrogram and as we will explain in section 4, our ranking logic prefers more specific patterns as they are more likely to produce natural, human-level fixes by taking into account more context. We found that the most meaningful patterns can often be found in inner nodes of the resulting dendrograms while patterns closer to the root are too general. In section 5 we evaluate the impact of our hierarchical approach on Getafix's accuracy of predicting human bug fixes.

3.3 Mining additional context

We already described how we mine not only bare fix patterns, but also include unmodified nodes which help making fix patterns more specific (see section 3.1). More specific fix patterns match in less spots and are hence more likely to produce valid fixes. However, including unmodified nodes can also bind otherwise unbound holes in the *after* part of a pattern, paving the way for otherwise inapplicable fix patterns. Figure 3 gives an example of an unbound edit that becomes bound thanks to additional context: Fix pattern (a) is *open* since `h0` is unbound. Fix patterns (b) and (c) are closed since `h0` is bound by an unmodified node in the *before* part and via *error context*, respectively. We explain the latter in the following paragraph.

Error context. We also allow associating edits with bug reports (static analysis tool, type error, lint message) that prompted the change in the first place, which allows us to learn how edit patterns relate to the corresponding bug reports. For instance, the expression blamed for potential `NullPointerExceptions` is shown as âĂIJerrorVarâĂİ in Figure 2 and participates in anti-unification, ending up as hole `h0`. This allows us to later substitute a blamed expression into `h0` when presented with a new bug report, making the fix pattern more specific. As with unmodified nodes, this can also bind otherwise unbound holes in the *after* part, which is shown in Figure 3(c).

Line distance distribution, to support ranking. Despite the previously mentioned measures to make fix patterns more specific, it is possible that a fix pattern matches at multiple spots within the same file. When fixing a bug, Getafix is usually provided with the location (line/column) of the bug report, so a first heuristic could be to always pick the match that is closest to the reported bug. However, this heuristic

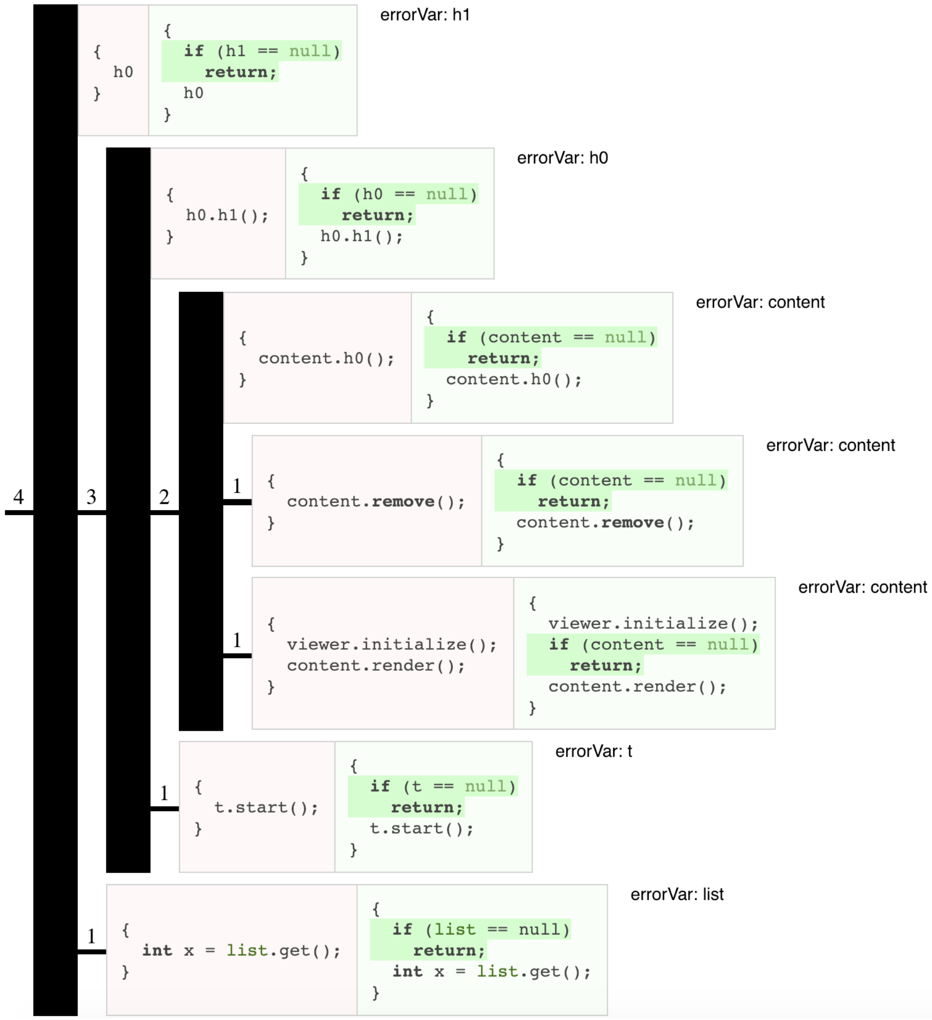
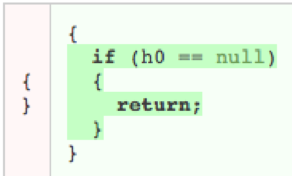


Figure 2. Dendrogram showing concrete edits merging into more abstract edit patterns. Each row shows an edit pattern - the *before* on the left and the *after* on the right, along with some metadata. The vertical black bars correspond to levels in the hierarchy, and the edit pattern at the top of each black bar is the pattern obtained by anti-unification of all the other edits belonging to that level in the hierarchy. The other edits are connected by the smaller, thin black lines.

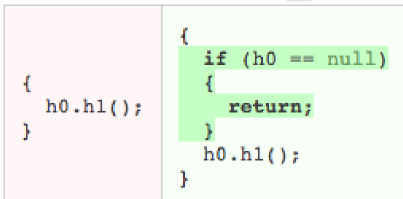
does not reflect reality in some cases, overall we observed fixes that happen:

- exactly in the reported line
- close, but never below the reported line
- arbitrarily far away above the reported line
- arbitrarily far away, above or below the reported line

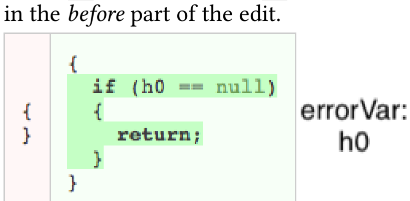
Thus, distinctive differences are whether proximity is important or not and whether fixes usually happen above/below the reported line. We hence model the distribution of expected relative distance between reported line and fix as two geometric distributions, one representing distance above, one representing distance below. Additionally, we store the overall above:below ratio. Examples of such distributions are shown in Figure 4. During hierarchical clustering, we initialize leaf nodes (concrete edits) by setting the above:below ratio to 0, 1 or 0.5 if the human fix happened above, below, or on the same line, respectively. We initialize the geometric distributions as ones having their expected value at exactly the observed distance. When combining nodes, we statistically combine the distributions and above:below ratios, giving us a representative overall distribution that can be used to compare different matches. Note that such distributions are also helpful to rank between matches of different *patterns*. For example we should prefer using the early return pattern 10 lines above the reported line over applying an inline fix 3 lines above the reported line. The specifics of this ranking are described in section 4.



(a) An example of a learned pattern that has an unbound hole. It is unclear from the pattern what to insert as `h0`.



(b) An example of a bound version of the pattern in Figure 3a. The previously unbound `h0` is bound because `h0` also appears in the *before* part of the edit.



(c) An example of a bound version of the pattern in Figure 3a. The previously unbound `h0` is bound by the variable mentioned in the bug report being addressed.

Figure 3. An example of an unbound edit and two ways which additional context can create a bound version of the edit.

4 Instantiation and Ranking

The final step takes buggy source code and fix patterns from the pattern mining step and produces patched versions of the source code. There are typically many fix patterns to choose from, so a challenge we have to address in this step is selecting the correct pattern to fix a particular bug. If the pattern applies in several locations, Getafix must also select the right match. The higher the right match is ranked, the less reliant the system is on the validation step later, thus saving time. The following examples illustrate our general approach and how we address this challenge.

Example 1. Consider the pattern we mined above: `h0.h1();` $\rightarrowtail$ `if (h0 == null) return; h0.h1();`

We briefly explain the steps to produce a patch on previously unseen code as shown in Figure 5. Getafix creates a patch using the following steps

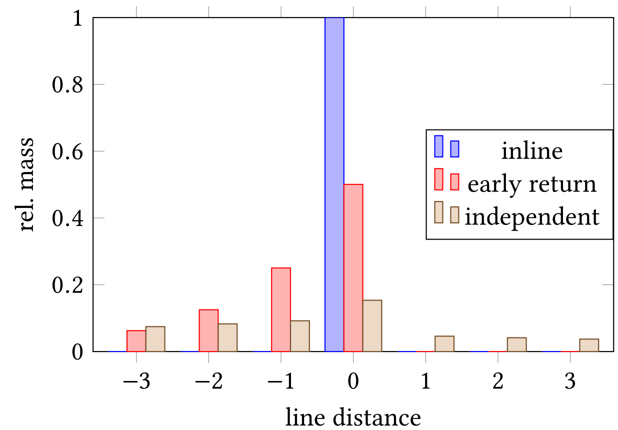


Figure 4. Example line distance distributions: blue represents inline fixes, red could represent the early return pattern, yellow could represent an independent pattern like modifying a field declaration responsible for a bug reported in some line where the field is used.

```
void onDestroyView() {
  mListView.clearListeners();
  mListView = null;
}

h0 ↦ mListView
h1 ↦ clearListeners
```

$\rightarrowtail$

```
void onDestroyView() {
  if (mListView == null)
    return;
  mListView.clearListeners();
  mListView = null;
}
```

Figure 5. Example patch.

1. Find sub-AST matching the *before* part: `mListView.clearListeners();`
2. Instantiate holes `h0` and `h1`
3. Replace sub-AST with instantiated *after* part

In practice, the pattern without unmodified nodes (like Figure 3c) will also appear higher up in the dendrogram, resulting from anti-unifying the pattern with unmodified node `h0.h1();` with a pattern inserting the return in front of a different statement. This pattern applies in unintended places, such as after `mListView.clearListeners();` or even after `mListView = null;` in the code of Figure 5. The next example illustrates how Getafix deals with patterns that seem to apply in too many places.

Example 2.
Consider the pattern `h0.h1()` $\rightarrowtail$ `h0!=null && h0.h1()`. Typically, this pattern would get mined from fixes within `if` conditions or `return` expressions, so we would expect it to be applied there as well. But it also matches in other situations, such as the call statement `mListView.clearListeners();` (see Figure 5). Getafix's ranking strategy tries to estimate the likelihood that a pattern is indeed a fix and also that it is the most likely fix for a given context, reducing the dependency on validation.

The above pattern will compete with other patterns, such as the more specific `if (h0.h1()) { ... }` $\rightarrowtail$ `if (h0!=null && h0.h1()) { ... }` or the pattern from Example 1, which applies only to call statements rather than expressions. More specific patterns match in fewer places and are thus considered to be more specialized for the situation, so Getafix ranks them higher. Next, we formalize the optimization problem behind ranking and describe the approximation we implemented.

Ranking. Since patterns for different bug types do not compete, we can focus on a single arbitrary bug type t. Let $\mathcal{B}$ denote the sets of bug instances of type t and $\mathcal{P}$ the set of available fix patterns for t. A bug instance $b \in \mathcal{B}$ includes its location (file and line number), so we can write $\langle p, b, z\rangle \in \mathcal{P} \times \mathcal{B} \times \mathbb{Z}$ for the action of patching b by applying p, z lines relative to b. So z is an offset that would for example be *negative* or zero in case of a successful early return fix, which should happen *above* the bug.

Example 4.1 (Applicable patches)**.** Let **NPE** be the NullPointerException bug instance fixed in Figure 5 and **ERvoid** the early return pattern for NullPointerException bugs in `void`-returning functions. Then $\langle \mathbf{ERvoid}, \mathbf{NPE}, 0\rangle$ is the patch shown in Figure 5. z denotes the delta between bug line in *before* and first modified line and is hence 0. Patches $\langle \mathbf{ERvoid}, \mathbf{NPE}, 1\rangle$ and $\langle \mathbf{ERvoid}, \mathbf{NPE}, 2\rangle$ are both applicable, but would insert the if statement too late in the method. The patch $\langle \mathbf{ERvoid}, \mathbf{NPE}, -1\rangle$ is not applicable since pattern **ERvoid** does not match on that line.

This example shows that not all such tuples are automatically applicable patches. We can easily decide whether a tuple is an applicable patch by *trying* to match the pattern's *before* at the desired spot. This test yields a predicate $\mathsf{match} \subseteq \mathcal{P} \times \mathcal{B} \times \mathbb{Z}$. Whether such an applicable patch will also *fix* the bug in question is captured by predicate $\mathsf{fix} \subseteq \mathsf{match}$ and can be decided by a validation step, which can usually be realized using the same tool that provided signal for the training data: static analysis, type checker, linter. Recall that due to limited time budget we do not want to rely on this validation step for ranking or candidate selection. At most, we validate a single candidate patch Getafix ranks highest, so fix is not at our disposal now. Yet another refinement is whether a fix not only passes validation, but is also the fix a *human* developer would perform, captured by $\mathsf{hfix} \subseteq \mathsf{fix}$. This predicate is generally undecidable, but our training data H, past *human fixes*, represents a lower bound for hfix. The overall relationship between predicates and training data can be summarized as

$$H \subseteq \mathsf{hfix} \subseteq \mathsf{fix} \subseteq \mathsf{match} \subseteq \mathcal{P} \times \mathcal{B} \times \mathbb{Z}$$

and is exemplified in Figure 6.

Given a specific $b \in \mathcal{B}$, Getafix will first enumerate fix candidates $C = \{ \langle p, b, z\rangle \in \mathcal{P} \times \{b\} \times \mathbb{Z}, \mathsf{match}(p, b, z) \}$. The ranking problem then is to estimate which of the applicable patches is the one most likely to resemble the human fix, i.e. find the p and z that maximize the probability that $\mathsf{hfix}(p, b, z)$ holds, based on our training data H. As H has no knowledge of b (a never seen before bug instance of type t), we generalize it away for our estimation. Since C contains only patterns known to match, we want to maximize

$$\mathbb{P}\big(\mathsf{hfix}(p, z) \mid \mathsf{match}(p, z)\big)$$

where $\mathsf{hfix}(p, z)$ is the event that applying p, z away from some previously unseen bug instance is what the human fix would look like. Analogously, $\mathsf{match}(p, z)$ is the event that p matches in some previously unseen line of code, z lines away from the bug instance. Knowing that our training data H records how to fix bugs $B_H = \{ b \in \mathcal{B} \mid \exists p, z. \langle p, b, z\rangle \in H \}$, we can approximate as follows:

$$\begin{aligned} &\mathbb{P}\big(\mathsf{hfix}(p, z) \mid \mathsf{match}(p, z)\big) \\ \approx\ &\frac{|\{ b \in B_H \mid \mathsf{hfix}(p, b, z) \}|}{|\{ b \in B_H \mid \mathsf{match}(p, b, z) \}|} \\ =\ &\frac{|H \cap (\{p\} \times \mathcal{B} \times \{z\})|}{|\{ b \in B_H \mid \mathsf{match}(p, b, z) \}|} \end{aligned}$$

We rewrite this into a product that can be approximated using the dendrogram:

$$\begin{aligned} &\frac{|H \cap (\{p\} \times \mathcal{B} \times \{z\})|}{|\{ b \in B_H \mid \mathsf{match}(p, b, z) \}|} \\ =\ &\frac{|H \cap (\{p\} \times \mathcal{B} \times \{z\})|}{|H \cap (\{p\} \times \mathcal{B} \times \mathbb{Z})|} \\ *\ &\frac{|H \cap (\{p\} \times \mathcal{B} \times \mathbb{Z})|}{|H|} \\ *\ &\frac{|H|}{|\{ b \in B_H \mid \mathsf{match}(p, b, z) \}|} \end{aligned}$$

We approximate each factor as follows:

- $\frac{|H \cap (\{p\} \times \mathcal{B} \times \{z\})|}{|H \cap (\{p\} \times \mathcal{B} \times \mathbb{Z})|}$ denotes: Out of the bugs fixed using p, what fraction was fixed by applying it z lines away. We model this using the *line distance distribution* for p (see section 3.3) and simply retrieve the mass at z. This factor favors patches happening in *similar proximity* to the bug as past fixes.
- $\frac{|H \cap (\{p\} \times \mathcal{B} \times \mathbb{Z})|}{|H|}$ is the fraction of bugs that was fixed using p, which is the relative multiplicity of p in the dendrogram. This factor favors patterns that were used *more frequently* in the past.
- $\frac{|H|}{|\{ b \in B_H \mid \mathsf{match}(p, b, z) \}|}$ equals $\frac{|B_H|}{|\{ b \in B_H \mid \mathsf{match}(p, b, z) \}|}$ due to $|H| = |B_H|$ (H records precisely one fix per bug). This is the *inverse* of the fraction of times that p matches z lines away from b (whether it is a fix or not). Note that whether p matches or not depends exclusively on the programming constructs used around that line (does the *before* part match or not). We assume that the expected distribution of programming constructs

Let p be the pattern "wrap statement in if" which is applicable to any line containing (the beginning of) a statement. Bug b is reported at statement `x.f();` (where $z = 0$) and we assume wrapping precisely that would be the human fix.

z	Code	match	fix	hfix	Comment
-1	`while (...) {`	✓	✓	✗	Can be wrapped and would make static analysis succeed, but we care about side-effect
0	`x.f();`	✓	✓	✓	Can be wrapped and makes both static analysis and us happy
1	`importantSideEffect();`	✓	✗	✗	Can be wrapped but static analysis fails
2	`}`	✗	✗	✗	Cannot be wrapped, no statement (begins) in this line

Figure 6. Example demonstrating relationship between predicates relevant for ranking.

does not significantly depend on b and z, so we can instead think of the fraction of lines p matches in an average source file. Since the factor is the inverse of this fraction, it favors *more specialized* patterns since they match more rarely. This could be approximated using the training data not only to mine $\mathcal{P}$, but also to compute this fraction for each $p \in \mathcal{P}$, which could then be annotated into the dendrogram. Getafix however currently estimates this fraction by looking at the complexity of the *before* part of p.

Example 4.2 (Ranking among candidates)**.**
Assume we learnt the two patterns for fixing Null Method Call exceptions shown in Example 1.1:

nER Introduce *new* "early return" at beginning of block.
eER Enhance *existing* "early return" above blamed spot.

Furthermore we assume that:

- Both patterns are rooted at the "method declaration" level of an AST for better comparability.
- 12% of methods seen in the training data contained an early return block (an if statement with a return).
- 10% of human fixes were **eER**, 90% were **nER**. The 2% mismatch could be due to cases where the value returned from existing early return blocks was not the value required for the fix.
- Both patterns include the blamed statement `h0 h1 = errorVar.h2();` as unmodified node (see Example 1.1) which makes the patterns significantly more specific and enforces order (early return happens *above* blamed statement).
- The average distance between if statement (inserted or enhanced) and blamed line is 6 lines and the fix always happens above the blamed line.
- Since we are interested in ranking, we assume that **eER** actually applies in the method containing bug b.

Knowing the relative multiplicities of both patterns we estimate:

$$\frac{|H \cap (\{\mathbf{nER}\} \times \mathcal{B} \times \mathbb{Z})|}{|H|} = 0.9$$
$$\frac{|H \cap (\{\mathbf{eER}\} \times \mathcal{B} \times \mathbb{Z})|}{|H|} = 0.1$$

We assume that 5% of all lines contain a method declaration, 10% of these methods happen to contain the unmodified statement involving the blamed errorVar. Since **nER** is an insertion-only pattern it will match all those methods, while **eER** only matches 12% of those:

$$\frac{|\{\ b \in B_H \mid \mathsf{match}(\mathbf{nER}, b, z)\ \}|}{|B_H|} \approx 0.05 * 0.1 = 0.005$$
$$\frac{|\{\ b \in B_H \mid \mathsf{match}(\mathbf{eER}, b, z)\ \}|}{|B_H|} \approx 0.05 * 0.1 * 0.12 = 0.0006$$

The last component of our estimated fix probability depends on the distance z between bug b and patch (first line touched counts, which is the if statement). Since we modelled the line distance distribution (only the "above" half matters here) as a geometric distribution with mean 6, we estimate:

$$\frac{|H \cap (\{\mathbf{nER}\} \times \mathcal{B} \times \{z\})|}{|H \cap (\{\mathbf{nER}\} \times \mathcal{B} \times \mathbb{Z})|} \approx \frac{1}{7} * (\frac{6}{7})^{|z|}$$
$$\frac{|H \cap (\{\mathbf{eER}\} \times \mathcal{B} \times \{z\})|}{|H \cap (\{\mathbf{eER}\} \times \mathcal{B} \times \mathbb{Z})|} \approx \frac{1}{7} * (\frac{6}{7})^{|z|}$$

Note that $|z|$ for **nER** cannot be smaller than any $|z|$ for **eER** since the new if statement is always inserted at the beginning of the method while existing ones might be further down and hence closer to us. Overall we estimate:

$$\mathbb{P}\big(\mathsf{hfix}(\mathbf{nER}, z) \mid \mathsf{match}(\mathbf{nER}, z)\big) \approx 26 * (\frac{6}{7})^{|z|}$$

$$\mathbb{P}\big(\mathsf{hfix}(\mathbf{eER}, z) \mid \mathsf{match}(\mathbf{eER}, z)\big) \approx 238 * (\frac{6}{7})^{|z|}$$

The estimations are very large given that we assumed that the world only knows two possible fixes (leads to large $\frac{|H \cap (\{p\} \times \mathcal{B} \times \mathbb{Z})|}{|H|}$) while assuming a realistic complexity and variety of Java constructs (leads to large $\frac{|B_H|}{|\{\ b \in B_H \mid \mathsf{match}(p, b, z)\ \}|}$). This bias, however, only contributes a constant factor and is hence irrelevant for ranking in this case.

We conclude that Getafix would clearly favor **eER** due to its rarity, furthermore prioritizing closer preexisting early return blocks in case the method contains multiple.

Duplication prevention There is always a risk of patch creation attempting multiple fixes on the *same* location in code. Consider the following snippet:

```
void attachImages(@Nullable View v) {
  for (Image img : this.images) {
```

```
3     v.attach(img);
4   }
5   screen.add(v.construct());
6 }
```

Static analysis would warn about potential `NullPointerException` in lines 3 and 5. Assuming that patch creation only knows the pattern wrapping *bare* method call statements[5], the highest ranked match to fix both warnings will be wrapping line 3. To prevent line 3 to be wrapped twice, we bail out before touching any line of code for a second time.

A similar scenario can arise if the line we *can* apply a patch to *is* already safe:

```
1 void attachImage(@Nullable View v) {
2   if (v != null) {
3     v.attach(this.image);
4   }
5   screen.add(v.construct());
6 }
```

While static analysis now only warns in line 5, we again assume that patch creation only knows the wrap pattern matching line 3. To the human eye (and the static analysis tool used, which we can however not rerun for each candidate), line 3 is already protected. The best fully automatic, language-agnostic approximation to this intuition is checking whether one of the known fix patterns matches *in reverse* at the spot we intend to patch. If a code location can be "unfixed", patch creation assumes that it already is fixed and bails out. This is also highly relevant for preventing patterns that add an `import` from adding one that already exists.

5 Evaluation

Getafix is deployed in Facebook to automatically suggest fixes for null dereference bugs reported by Infer, our static analysis tool, as well as to suggest fixes for the null dereference-related crash bugs that Sapienz finds. It is also being used to resolve outstanding Infer bugs from the past. We evaluated Getafix both based on its ability to precisely predict human fixes and based on the impact of its deployment. At the end of this section we present a selection of (anonymized) fixes that were accepted.

Accuracy of predicting human fixes of Infer warnings. In one experiment, we compared Getafix-computed fixes with actual human-written past fixes for the same Infer Null Method Call[6] bugs. From the repository of the Facebook Android app we collected five months worth of file pairs containing human bug fixes. We dropped all file pairs representing a change touching more than four lines in order to reduce noise: Humans often combine a fix with other changes, which would both pollute training data and result

[5]This is a naive assumption, but for more sophisticated patterns there is always a more sophisticated counterexample

[6]https://fbinfer.com/docs/eradicate-warnings.html#ERADICATE_NULL_METHOD_CALL

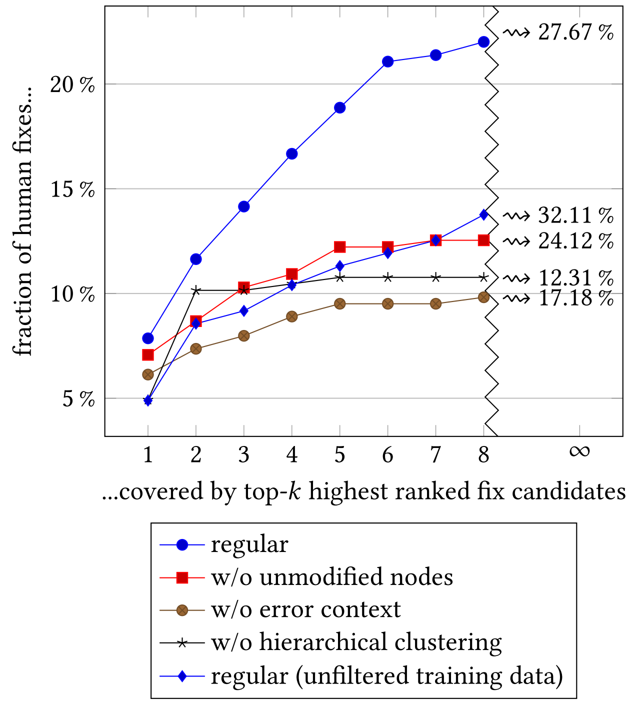


Figure 7. Accuracy of Getafix predicting human fixes for Infer Null Method Call bugs, compared to accuracy with certain features disabled.

in sub-optimal test data, since our goal is not to reproduce those other changes. As training and test data we took three and two months of that data, which resulted in 585 and 427 file pairs, respectively. We configured the pattern mining step to drop patterns representing less than 1% of the training set, which yielded 67 patterns.

Without further inspection or modification, all 67 patterns were then used to attempt reproducing each of the test pairs, i.e. applying our patch creation and ranking tool to the *before* file and checking if one of the candidate patches is the *after* file. This comparison only neglects empty lines but is otherwise precise, so even the addition of a comment by the human or different whitespace choices would be judged as a mismatch. *Our goal is to produce natural, human-like fixes*, so no mismatch is acceptable if a human is supposed to accept our fixes with a clear conscience.

Figure 7 shows the test results. We did not only use Getafix unchanged ("regular"), but also measured its accuracy with certain features turned off, in order to estimate the impact of innovations. It is not only important that the we reproduced the human fix with one of our candidates, it is also important that this matching candidate is ranked high. Therefore, we plot accuracy of the top-k highest ranked candidates for increasing k, showing us for example for $k = 1$ that in about 8% of cases the highest-ranked candidate matched the human fix, for $k = 3$ that the human fix is among the top three candidate in about 14% of cases, and for $k = \infty$ that the human fix was found at all in 27.67% of cases.

Without the inclusion of unmodified nodes into the mining and patching process, overall accuracy drops to 24.12% which is due to certain fix patterns no longer being closed (holes in the *after* were bound by unmodified holes in *before*). Overall ranking suffers even more significantly (over 9% drop in top-8 score vs under 4% drop overall) since unmodified nodes represent the "typical context" a fix usually appears in. Furthermore, patterns may now match in more spots of a source file, so there are more fix candidates to compete with. Without error context, even more patterns are no longer closed (holes in *after* were bound by something that could be parsed out of the Infer message). Again, patterns now also apply in more spots (if nodes were bound by both a hole in the *before* and the Infer message, they are now only bound by the *before* and will hence match whatever is found in the source file). The most drastic drop in accuracy can be observed without the hierarchical clustering approach and hence without the valuable inner nodes of the dendrogram. Instead, d-cap is used for partitioning and fix patterns are generalized iteratively as anti-unification with a new concrete edit still results in a closed pattern. This resulted in only 14 patterns that capture the overall shape of human fixes well, but are too general for well-informed ranking.

Inspecting the remaining 72% of human fixes that were not reproduced showed:

- Swapping out blamed expressions (e.g. `this.getList()` rather than `this.list` knowing that the getter contains a null check)
- Renaming of variables
- Plain removal of the buggy code
- Fixing a deeper root cause (e.g. `x != null && y.foo()` should always have been `y != null && y.foo()` while we suggested newly adding `y != null &&`)
- Syntactic variations of well-known patterns that just neither showed up in the training data nor got covered by generalizations during anti-unification

In a slight variation of the experiment we did not drop file pairs representing a change touching more than four lines from the training data (2488 file pairs; test data left unchanged). While the accuracy among all fix candidates rose to 32%, the additional noise resulted in worse accuracy of top ranked candidates.

Human influence on patterns can improve accuracy and ranking by dropping patterns considered negligible and by adding *interpolations* of existing patterns, rare variations of patterns that did not show up in the training data, but that can be mass-produced with domain knowledge about the programming language. In an earlier experiment we measured accuracy of patterns that were hand picked from the mining process and then extended with interpolations. The test set contained about 200 small of human fixes and we focussed only on the highest-ranked patch since this is also the one we would present in practice. In about 25% of cases, our patch reproduced the human fix. In another 25% of cases, our patch addressed the warning according to Infer (warning no longer reported), but not necessarily in exactly the way the human did.

One significant reason for the low accuracy among few top ranked candidates is that there are at least two fundamentally different ways to address a Null Method Call warning: Either one recognizes that the warning (which is essentially a type error of the null-safe type system Infer enforces) truly indicates the possibility that the blamed expression can be `null` at runtime or one is certain that for reasons inaccessible to the employed logic, the expression can in fact not be null (note that this imprecision of the static discipline is nothing uncommon and the reason casts exist). Depending on this, one will either change logic or control flow to behave reasonably in case of `null` or add an non-null assertion (acting like a cast) that convinces Infer about non-nullability. We found that there is no clear pattern that indicates which path is chosen by humans, presumably domain knowledge and code understanding is required for this. This may be an opportunity for deep learning based techniques. There are also numerous ways to proceed in both of these cases (e.g. which line to put the assertion on), which explains the steep incline in precision among top candidates.

Displaying auto-fix suggestions during code review. Getafix suggests fixes for issues reported by Infer during code review by leaving a comment containing a change suggestion that can be accepted or rejected with one click. We are interested in the impact of displaying auto-fix suggestions during code review. Metrics are:

- the number of fixes accepted (which results in time savings for not having to write and pushing changes)
- increase in overall fix rate when displaying fix suggestions
- reduction of time between introduction and resolution of bugs

For this deployment we manually selected and enhanced patterns with interpolations (see previous paragraph). For example, we dropped all patterns that add assertions to the code or that just remove code, given that these patterns produce a plethora of candidates that will be hard to rank amongst. Before displaying auto-fix suggestions we rerun Infer to ensure that previously reported warnings have disappeared.

Within 3 months, developers addressed around 250 Null Method Call warnings that we also showed a fix suggestion for. Figure 8 categorizes their reactions: In 106 of those cases, i.e. around 42% of the time, they directly accepted our suggestion. The remaining cases (fix not accepted) are sorted and colored roughly by their potential to be claimed by Getafix. In 10% of cases they wrote a fix that Getafix knows but did not rank highest and hence did not suggest. Better ranking

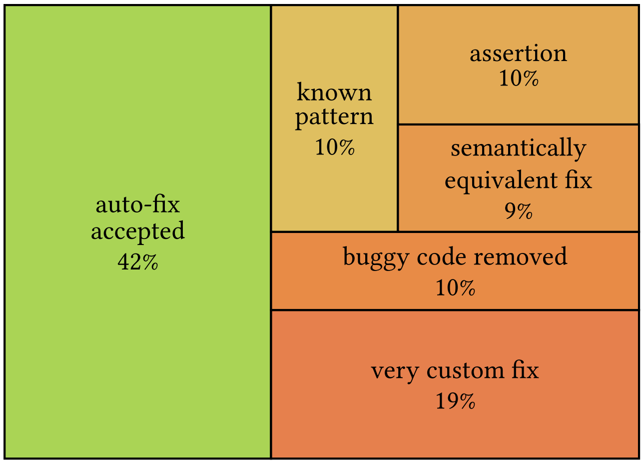


Figure 8. Reactions to 250 Null Method Call warnings that come with a fix suggestion.

could hence predict more of these cases. Another 10% of the time the developer instead added an assertion. As indicated earlier, we have intentionally not rolled out such patterns so far. Around 9% of the time a semantically equivalent fix (that we have no pattern for) was written, e.g. for a bug in an `else` block, turn it into an `else if` rather than creating a new `if`. More specialized patterns through more training data or manual interpolations will help. In around 10% of cases the issue was resolved by just removing the code that contained the warning. The remaining human fixes were mostly very custom (similar to not reproducible cases mentioned in previous paragraph) and while they can generally be expressed as a pattern (note that even a concrete edit is a valid but very specific pattern), we do not see sufficient value in doing so.

Getafix has also been suggesting autofixes for Infer's Field Not Nullable and Return Not Nullable warnings. Within 3 months, displaying an auto-fix next to a warning resulted in 4% and 12% higher fix rate, respectively. This amounts to around 80 additionally fixed warnings (across both warning types) per month. The acceptance rate of our autofixes is at around 60% for both types. We observed that Return Not Nullable warnings are usually fixed around twice as fast if an auto-fix is displayed. No comparably significant speedup for Field Not Nullable warnings could be found so far.

Selection of accepted fixes. The following anonymized fixes were suggested in the code review portal and accepted by the author. Even though the addressed bug from Infer is the same each time (Null Method Call[7] warnings, which indicates the risk of a `NullPointerException` being thrown), each fix is unique. Notice that the fixes are indistinguishable from the kinds developers typically make.

[7] https://fbinfer.com/docs/eradicate-warnings.html#ERADICATE_NULL_METHOD_CALL

Example 5.1 (Extend existing if-condition conjunctively).

```
// ...
Action a = actions.getActionForKey(key);
if (a == null && a.shouldExecute()) {
  a.execute();
}
// ...
```

Example 5.2 (Extend existing if-condition conjunctively).

```
// ...
if (key != 0 && mThread != null) {
  mThread.cancel(key);
} else {
  ThreadPool.tryCancel(key);
}
// ...
```

Example 5.3 (Extend existing if-condition disjunctively).

```
boolean process() {
  if (this.lazyProvider == null || manager.get() == null) {
    return false;
  }
  Provider p = this.lazyProvider.get();
  // ...
}
```

Example 5.4 (Protect call with conditional).

```
@Override
public void onClose() {
  final Window w = mWindow != null ? mWindow.get() : null;
  if (w != null && w.getProc().isActive()) {
    w.getProc().removeListeners();
  }
}
```

Example 5.5 (Add an early return).

```
void updateView() {
  // ...
  View v = ViewUtil.findViewWithName(name);
  if (v == null) {
    return;
  }
  data.send(((UpdatableView) v).getContentMgt(), "UPDATE");
  // ...
}
```

Example 5.6 (Protect call with new if statement).

```
// ...
final Button b = getButton();
if (b != null) {
  Styles.setGreyOut(b.getStyleRef(), isDisabled);
}
// ...
```

Bulk-fixing of outstanding Infer warnings. In addition to suggesting fixes for new Infer bugs as they are introduced, we have also been using our same infrastructure to clean up the backlog of Infer bugs that made it past code review and into the codebase. We have cleaned up over two thousand Return Not Nullable[8] Infer bugs and over a thousand Field Not Nullable[9] Infer bugs as a part of this effort.

SapFix. Getafix also produces fixes for SapFix to address the crashes that Sapienz detects. Over the past months, Getafix provided about half of the fix candidates that SapFix uses and considers valid (all tests passed). Of all fix candidates Getafix provides to SapFix, about 80 percent pass all tests.

6 Related Work

In contrast to genetic approaches like GenProg [3], our work focusses solely on trying to learn patterns from past fixes. There is no attempt to find generic solutions from any sort of ingredient space, or by generically mutating the code. While Genesis [8] follows a similar approach to learning transformations between *before* and *after* ASTs, it contains generators in the learned templates which increase the size of the search space. This allows multiple of our edit patterns to be represented by a single one of of their patterns, but the tradeoff is that more time is spent exploring that space of patches at generation time. Our patterns represent instantiated instances of their templates with collections of ingredients that make sense, such as ingredients `==` and `||` to create a pattern that inserts `h0 == null ||` into an existing conditional that does an early return. The specificity of our patterns means we do not have to explore the space of possible fixes and can focus on validating the patches that are most likely to be correct. This is necessary to achieve one of our design goals, mentioned in section 1, of relying on validation as little as possible.

SketchFix [4] addresses the goal to compile and validate as little as possible very differently, by starting with fix sketches that are then refined *at runtime*, during test execution, i.e. by reacting to runtime behavior like exceptions and repeatedly backtracking test execution until everything passes.

Kim et al. [5] automatically group a large number human-written patches by similarity, producing a manageable number of groups that can each potentially be generalized into a fix pattern. Inspection of these groups and creation of fix patterns is done manually. Getafix aims to learn fix patterns fully automatically.

History driven program repair as investigated by Le et al. [7] influenced our work, the overall diffing/mining pipeline is similar. We believe our approach is more generic in that it does not represent fix patterns using a set of specific mutation operators (like "insert statement"), but instead operates directly on raw ASTs. ((For the mining part we developed a hierarchical clustering algorithm that is able to efficiently find patterns among order of 100,000 concrete edits. Furthermore, we focused on mining additional context (see section 3.3) that enables us to select and rank among thousands of fix patterns effectively or detect whether applying multiple patterns at once is appropriate.))

Refazer [15] is a technique that uses PROSE [13] and a DSL of program transformations to learn fixes and repetitive edits. Revisar [16] influenced Getafix, it also uses anti-unification [6] to derive fix patterns from concrete edits. However, they use a greedy clustering algorithm while we developed a hierarchical one resulting in a dendrogram of fix patterns.

Soto et al. [18] make a point about fix patterns being different for different programming languages and analyze the structure of typical patterns in Java. Martinez and Monperrus [10][11][12] also performed extensive AST analysis on Java repositories to statistically analyze code change patterns, which can guide program repair. Soto and Goues [17] propose a probabilistic model-based approach for Java which produces higher quality patches more likely and also addresses the multi-pattern fix problem. It is our goal that Getafix, not having any expectation or even notion about semantics, will automatically learn all these structural properties per language.

CapGen [19] uses different forms context of to influence selection of mutation operators and fixing ingredients. Their "genealogy context" looks at AST neighborhood to prioritize ingredients based on the neighborhood one would typically find them in. We implicitly realize a similar prioritization using our clustering and ranking strategy: During clustering, both siblings (unmodified nodes) and ancestors are retained as much as possible, resulting in a spectrum of patterns with varying amounts of detailed context and different relative multiplicity. Fix pattern selection will filter from this spectrum all matching patterns, the ranking strategy (fix probability estimation) then naturally prefers patterns with more context, in combination with higher multiplicity.

7 Future Work

Fix interpolation. Seemingly insignificant differences in source code can result in significant changes in the AST representation. As a result, fix patterns learnt by Getafix may be very sensitive to subtle variations of buggy code which it could otherwise fix. For example, assume that Getafix knows the fix pattern wrapping call statement `h0.h1();` with a null-check for `h0` if it is the blamed variable. The fix pattern's *before* AST knows the method call expression `h0.h1()` as the child of an expression statement. This pattern will not match a statement `g(x.f());` which has `x.f()` as the *grandchild* of an expression statement (see Example 5.6).

[8] https://fbinfer.com/docs/eradicate-warnings.html#ERADICATE_RETURN_NOT_NULLABLE

[9] https://fbinfer.com/docs/eradicate-warnings.html#ERADICATE_FIELD_NOT_NULLABLE

A fix pattern matching this example statement would be a slight variation of the original pattern. We call such slight variations *interpolations* of the original pattern. Further AST variations like cast expressions (e.g. `((Foo)x).f()` or `(Foo)x.f()`) lead to combinatorial explosion of interpolations required to reliably match all statements containing `h0.h1()`.

Getafix will learn any number of interpolations automatically when presented with a sufficient amount of samples in the training data. However, it is possible that a fraction of bugs seen in the wild has an AST structure that did not occur in the training data. One could manually assist Getafix by enumerating all fix pattern interpolations up to a desired maximum complexity (which requires language-specific knowledge) . We have also considered introducing a *descendant* relation in fix patterns in order to support representing above mentioned interpolations as a single fix pattern. We did not investigate this idea further, for example it is unclear how/when the pattern mining step would opt for the descendant relation.

Machine learning. A variety of components of our pipeline could potentially benefit from integrating machine learning techniques: For favoring special patterns for special circumstances, we currently rely on a combination of strong *before* parts of a pattern and statistical reasoning to favour such strong patterns. For example, this will ensure early return patterns that match on the return type (like returning `false` if the return type was `boolean`) are selected among more generic patterns. This is a very strict and purely syntactic approach, which might not pick up on more subtle hints that could for example decide between early return patterns and wrapping patterns. We have already experimented with training a classifier, hoping to be more sensitive to such subtleties.

Pradel and Sen [14] showed promising results in purely name-based bug detection. It is likely that using a similar technique would benefit our learning and ranking phase, at the very least one could use word embeddings to capture the "flavor" of values that were substituted by holes. For example, we have seen name-based indicators for when humans tend to fix Null Method Call warnings by wrapping the call with an if: Method being wrapped are often logging calls, destruction (`release`, `close`), events (`on...`) or GUI updates (`focus`, `setDefault`), i.e. side effects that should have no further consequence on the control flow at the call site. If the subject is `null` there is legitimately nothing to be done, rather than, for example, coming up with an alternative value for the subject. Another potential application would be mimicking human intuition about the concrete return value to pick in case of an early return, for example `-1` for `indexOf`-like functions or `null` vs `""` for string functions. We suspect that a name-based approach would be able to pick up those clues.

Getafix so far cannot learn change patterns within individual token values. For example appending suffix `Async` to a method name could be represented by a pattern `h0` $\rightarrowtail$ `<h0 + "Async">` rather than `h0` $\rightarrowtail$ `h1` (which we would currently learn from two concrete rename edits `get` $\rightarrowtail$ `getAsync` and `put` $\rightarrowtail$ `putAsync`). Getafix can represent either constant values or the act of copying a token from the *before* tree precisely (via holes), but it can not perform simple computations on existing values to derive new values. A strategy could be to use PROSE [13] during anti-unification: When currently we would anti-unify a pair of tokens in the *after* trees with a new unbound hole (if no existing hole matches exactly and can hence be reused), we could instead attempt finding a function that can compute the token values based on existing holes.

Open patterns. Getafix generally does not attempt to apply "open" fix patterns, i.e. if a holes in the *after* part is bound neither by the *before* part nor the bug report. With an appropriate method to instantiate the unbound holes, these fix patterns may prove useful as well. For example, one could attempt instantiation using ingredients as done for example by [19]. In a more interactive context, for instance when suggesting fixes in an IDE, one could also let the developer choose an instantiation.

At Facebook. Getafix has helped us advance toward our goal of letting computers take care of routine bug-fixing work. As we continue to refine our testing and verification tools, we expect Getafix will be able to prevent a larger portion of postdeployment failures.

We note that the fix patterns Getafix mines need not come only in response to Infer-related fixes. Indeed, they can also come from fixes made during or in response to manual code review. This additional source of fix patterns opens up the exciting possibility of automating repetitive code reviews. In other words, a bug that has been flagged and remediated across the codebase multiple times in the past can be flagged automatically in a future code commit âĂŤ without a human needing to do it.

Getafix is part of our overall effort to build intelligent tools that rely on statistical analysis of large code corpora and the associated metadata. Such tools have the potential to improve all aspects of the software development life cycle, including code discovery, code quality, and operational efficiency. The insights we gain from Getafix will help us in building out and deploying additional tools in this space.

Acknowledgments

We thank fellow Facebook colleagues for their contributions and help (chronologically): Eric Lippert implemented a custom version of the GumTree tree differencing algorithm and investigated an edit script graph approach to learning code change patterns. Jeremy Dubreil from the Infer team worked with us to integrate Infer fix suggestions into the existing Infer bug reporting workflow. Alexandru Marginean from the Sapienz team integrated Getafix into the SapFix pipeline as one of its strategies to create fix candidates. Austin Luo applied Getafix to Hack (programming language), working on learning lint patterns from past change suggestions made during code review. Waruna Yapa applied Getafix to Objective-C, working on learning fixes for further classes of Infer warnings (for example "Bad Pointer Comparison"). Vijayaraghavan Murali created and trained the classifier mentioned in section 7 that provides an alternative approach for ranking between fix patterns.

References

[1] Jean-Paul Benzécri. 1982. Construction d'une classification ascendante hiérarchique par la recherche en chaîne des voisins réciproques. *Cahiers de l'analyse des données* 7, 2 (1982), 209–218. http://www.numdam.org/item/CAD_1982__7_2_209_0

[2] Jean-Rémy Falleri, Floréal Morandat, Xavier Blanc, Matias Martinez, and Martin Monperrus. 2014. Fine-grained and Accurate Source Code Differencing. In *Proceedings of the International Conference on Automated Software Engineering*. Västeras, Sweden, 313–324. https://doi.org/10.1145/2642937.2642982 update for oadoi on Nov 02 2018.

[3] Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. 2012. GenProg: A Generic Method for Automatic Software Repair. *IEEE Transactions on Software Engineering* 38 (2012), 54–72.

[4] Jinru Hua, Mengshi Zhang, Kaiyuan Wang, and Sarfraz Khurshid. 2018. Towards Practical Program Repair with On-demand Candidate Generation. *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)* (2018), 12–23.

[5] Dongsun Kim, Jaechang Nam, Jaewoo Song, and Sunghun Kim. 2013. Automatic patch generation learned from human-written patches. In *Proceedings of the 2013 International Conference on Software Engineering*. IEEE Press, 802–811.

[6] Temur Kutsia, Jordi Levy, and Mateu Villaret. 2014. Anti-unification for Unranked Terms and Hedges. *Journal of Automated Reasoning* 52, 2 (01 Feb 2014), 155–190. https://doi.org/10.1007/s10817-013-9285-6

[7] Xuan-Bach D. Le, David Lo, and Claire Le Goues. 2016. History Driven Program Repair. *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)* 1 (2016), 213–224.

[8] Fan Long, Peter Amidon, and Martin Rinard. 2017. Automatic inference of code transforms for patch generation. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. ACM, 727–739.

[9] Fan Long and Martin Rinard. 2016. Automatic Patch Generation by Learning Correct Code. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '16)*. ACM, New York, NY, USA, 298–312. https://doi.org/10.1145/2837614.2837617

[10] Matias Martinez and Martin Monperrus. 2012. *Mining repair actions for guiding automated program fixing*. Ph.D. Dissertation. Inria.

[11] Matias Martinez and Martin Monperrus. 2015. Mining Software Repair Models for Reasoning on the Search Space of Automated Program Fixing. *Empirical Softw. Engg.* 20, 1 (Feb. 2015), 176–205. https://doi.org/10.1007/s10664-013-9282-8

[12] Matias Martinez and Martin Monperrus. 2018. Coming: a Tool for Mining Change Pattern Instances from Git Commits. arXiv:arXiv:1810.08532

[13] Oleksandr Polozov and Sumit Gulwani. 2015. FlashMeta: A Framework for Inductive Program Synthesis. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2015)*. ACM, New York, NY, USA, 107–126. https://doi.org/10.1145/2814270.2814310

[14] Michael Pradel and Koushik Sen. 2018. DeepBugs: A Learning Approach to Name-based Bug Detection. *CoRR* abs/1805.11683 (2018). arXiv:1805.11683 http://arxiv.org/abs/1805.11683

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

[16] Reudismam Rolim, Gustavo Soares, Rohit Gheyi, and Loris D'Antoni. 2018. Learning Quick Fixes from Code Repositories. *CoRR* abs/1803.03806 (2018). arXiv:1803.03806 http://arxiv.org/abs/1803.03806

[17] M. Soto and C. Le Goues. 2018. Using a probabilistic model to predict bug fixes. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, Vol. 00. 221–231. https://doi.org/10.1109/SANER.2018.8330211

[18] Mauricio Soto, Ferdian Thung, Chu-Pan Wong, Claire Le Goues, and David Lo. 2016. A Deeper Look into Bug Fixes: Patterns, Replacements, Deletions, and Additions. In *Proceedings of the 13th International Conference on Mining Software Repositories (MSR '16)*. ACM, New York, NY, USA, 512–515. https://doi.org/10.1145/2901739.2903495

[19] Ming Wen, Junjie Chen, Rongxin Wu, Dan Hao, and Shing-Chi Cheung. 2018. Context-Aware Patch Generation for Better Automated Program Repair. *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)* (2018), 1–11.