Getafix: Learning to Fix Bugs Automatically

JOHANNES BADER, Facebook, USA
ANDREW SCOTT, Facebook, USA
MICHAEL PRADEL*, Facebook, USA
SATISH CHANDRA, Facebook, USA

Static analyzers help find bugs early by warning about recurring bug categories. While fixing these bugs still remains a mostly manual task in practice, we observe that fixes for a specific bug category often are repetitive. This paper addresses the problem of automatically fixing instances of common bugs by learning from past fixes. We present Getafix, an approach that produces human-like fixes while being fast enough to suggest fixes in time proportional to the amount of time needed to obtain static analysis results in the first place.

Getafix is based on a novel hierarchical clustering algorithm that summarizes fix patterns into a hierarchy ranging from general to specific patterns. Instead of an expensive exploration of a potentially large space of candidate fixes, Getafix uses a simple yet effective ranking technique that uses the context of a code change to select the most appropriate fix for a given bug.

Our evaluation applies Getafix to 1,268 bug fixes for six bug categories reported by popular static analyzers for Java, including null dereferences, incorrect API calls, and misuses of particular language constructs. The approach predicts exactly the human-written fix as the top-most suggestion between 12% and 91% of the time, depending on the bug category. The top-5 suggestions contain fixes for 526 of the 1,268 bugs. Moreover, we report on deploying the approach within Facebook, where it contributes to the reliability of software used by billions of people. To the best of our knowledge, Getafix is the first industrially-deployed automated bug-fixing tool that learns fix patterns from past, human-written fixes to produce human-like fixes.

CCS Concepts: • **Software and its engineering → Software testing and debugging**.

Additional Key Words and Phrases: Automated program repair, Patch generation, Code transform

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

1 INTRODUCTION

Modern production code bases 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 code bases. While finding bugs early via static analysis is helpful, the problem of fixing these bugs still remains a mostly manual task in practice, hampering the adoption of static analysis tools [Christakis and Bird 2016].

Most static analyzers look for instances of common bug categories, such as potential null dereferences, incorrect usages of popular APIs, or misuses of particular language constructs. We

*This work was done in part while this author was a visiting scientist at Facebook.

Authors' addresses: Johannes Bader, Facebook, USA, jobader@fb.com; Andrew Scott, Facebook, USA, andrewscott@fb.com; Michael Pradel, Facebook, USA, michael@binaervarianz.de; Satish Chandra, Facebook, USA, satch@fb.com.

2475-1421/2019/10-ART159
https://doi.org/10.1145/3360585

Extend existing if condition conjunctively:

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

Add an early return:

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

Protect call with conditional expression:

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

Fig. 1. Different real-world fixes of potential null dereference bugs.

observe that fixes for a specific bug category often resemble each other: there is a pattern to them. That is, past human fixes of the same bug category may offer insights into how future instances of the bug category should be fixed. Given this observation, can we automate fixing bugs identified by learning from past fixes?

This paper addresses the problem of automatically fixing instances of common bug categories by learning from past fixes. We assume two inputs: (1) A set of changes that fix a specific kind of bug, e.g., from the version history of a code base. These changes serve as training data to learn fix patterns from. (2) A piece of code with a static analysis warning that we want to fix. Given only these two inputs, the problem is to predict a fix that addresses the static analysis warning in a way similar or equal to what a human developer would do. By automating the fix generation and leaving to the human only the final decision of whether to apply the fix, the overall effort spent on addressing bugs pointed out by static analyzers can be greatly reduced.

We focus on kinds of bugs that have non-trivial yet repetitive fixes. On the one end of the spectrum, there are bug categories that trivially imply a specific fix. For example, for a warning that suggests a field to be `final`, implementing an automated fix suggestion is straightforward. Such an auto-fix can be defined by the author of that rule in the static analyzer, without knowing the specific context in which the rule is applied; indeed, some of Error Prone [Aftandilian et al. 2012] rules come with auto-fixes. On the other end of the spectrum are bugs that require complex, application-specific fixes, such as an issue with a UI tab not displaying after a specific series of interactions from a user. Here we target bug categories in between these two extremes, where finding a fix is non-trivial, yet typical fixes belong to a set of recurring fix patterns. For such bug categories, there often exists more than one way to fix the problem, and the "right" way to address a specific instance of the bug category depends on the context, e.g., the code surrounding the static analysis warning.

As an example of a bug category targeted in this work, consider `NullPointerExceptions` – still one of the most prevailing bugs in Java and other languages. If a static analyzer warns about a potential null dereference, developers may fix the problem in various ways. Figure 1 shows three anonymized examples of fixes of null dereference bugs, which add a conjunct to an existing if condition, replace a call with a ternary operation, and add an early return, respectively. While all these fixes introduce some kind of null check, the exact fix heavily depends on the already existing code. Beyond these examples, there are even more ways of fixing null dereference bugs, e.g., by adding a new if statement or by extending an existing if condition disjunctively. Learning all these fix patterns and deciding which one to apply to a given piece of buggy code is a non-trivial problem.

Our work aims at automating bug fixing in large-scale, industrial software development. This setting leads to several interesting challenges to deal with:

- To reduce the human time spent on fixing a bug, the approach may propose only a *small number* of potential fixes, ideally only one fix.
- To make this fix acceptable to developers, the suggested fix should be *human-like*: very similar to or exactly the same as a fix a human developer would implement.
- To suggest fixes *quickly*, as well as to keep the computing resources required to find fixes within bounds, the approach cannot explore a large space of candidate fixes and validate each of them against a test suite or any other expensive validation routine.

Addressing these challenges, we present Getafix, an automated technique that learns recurring fix patterns for static analysis warnings and suggests fixes for future occurrences of the same bug category. Getafix produces *human-like* fixes, and does so *fast enough* (typically within 10 seconds) to offer a fix suggestion in roughly the same magnitude of time as a human developer waits for static analysis results anyway. In a nutshell, the approach consists of three main steps. First, it splits a given set of example fixes into AST-level edit steps. Second, it learns recurring fix patterns from these edit steps, based on a novel hierarchical, agglomerative clustering technique that produces a hierarchy of fix patterns ranging from very general to very specific fixes. Third, given a previously unseen bug to fix, Getafix finds suitable fix patterns, ranks all candidate fixes, and suggests the top-most fix(es) to the developer. As a check during the third step, Getafix validates each suggested fix against the static analyzer to ensure that the fix removes the warning. Note that the validation against the static analyzer is a one-time effort per fix, keeping the computational resources within reasonable bounds.

The Getafix approach and the constraints that motivated it differ from the assumptions made by automated program repair techniques [Goues et al. 2019], the most practical of which for large systems are *generate-and-validate* repair systems [Goues et al. 2012; Kim et al. 2013; Le et al. 2016].

(1) Because it must produce fixes quickly, Getafix neither generates, nor validates a large number of fix candidates. Getafix produces a ranked list of fix candidates based entirely on past fixes it knows about and on the context in which a fix is applied. This is done without *any* validation; Getafix's ranking reflects the confidence it has in the fix. It then offers to the developer one (or a few, configurable) top-ranked fix(es) after a validation step. Note that Getafix will not attempt to validate beyond the configured number of fixes - if none of the top ranked fixes are valid then no fix will be offered.
(2) In contrast to generate-and-validate repair systems that use a test suite as validation filter, Getafix uses the same static analyzer as the validation filter to make sure the warning goes away. This is because at this point in the development, we cannot assume that the code is ready to pass a test suite.
(3) Prior repair techniques rely on statistical fault localization (e.g., based on coverage of passing and failing test cases) that determines where in the code a bug should be fixed. Getafix avoids this step by exploiting the fact that static analyzers pinpoint the location of the bug. Consequently, it also avoids the need to try applying fix candidates at various potential fault locations.
(4) Finally, the goal that Getafix sets for itself is not merely a fix that makes its validation pass, but is as close as possible to what a human would do in that situation. (While a generic null check would suppress a null dereference warning, as Figure 1 shows, a human might choose a very specific kind of fix in a particular context.) By contrast, most program repair systems aim to produce a fix that is functionally correct, but may not match a human fix.

Another related line of work is on learning edit patterns, including potential bug fix patterns, from version histories [Brown et al. 2017; Rolim et al. 2017, 2018]. In contrast to Getafix, these techniques learn from *all* code changes, leaving the task of identifying interesting fix patterns to a

human. A key insight of our work is that learning from fixes for a specific static analysis warning avoids this human effort.

We evaluate Getafix in two ways. One part of our evaluation applies the approach to a total of 1,268 bug fixes for six kinds of warnings reported by two publicly available and widely used static analyzers for Java. The bug categories include potential null dereferences, incorrect uses of Java's reference equality, and common API misuses. After learning fix patterns from several dozens to several hundreds of examples, Getafix predicts exactly the human fix as the top-most suggestion for 12% to 91% of all fixes, depending on the bug category. In a setting where developers are willing to inspect up to five fix suggestions, the percentage of correctly predicted fixes even ranges between 19% and 92%, containing fixes for 526 of the 1,268 bugs. Because these results indicate how often the predicted fix exactly matches the human fix, as opposed to producing any human-acceptable fix, these results provide a lower bound of the effectiveness of Getafix.

The other part of our evaluation deploys Getafix to production at Facebook, where it now contributes to the stability of apps used by billions of people. At Facebook, Getafix currently suggests fixes for bugs found by Infer [Calcagno et al. 2015][1], a static analysis tool that identifies issues, such as null dereferences, in Android and Java code. For example, the fixes in Figure 1 have been suggested by Getafix and accepted by developers at Facebook. We find that developers accept around 42% of all fixes suggested by Getafix, helping to save precious developer time by addressing bugs with a single click.

Contributions. This paper makes the following contributions:

- Fully automated suggestion of bug fixes for static analysis bug reports, computed in seconds, without requiring expensive search over a large space of candidate bug fixes.
- A novel clustering technique for discovering repetitive fix patterns in a way that is more general than a previous greedy approach and that preserves important context needed for applying fixes.
- Simple yet effective statistical ranking technique to select the best fix(es) to suggest, which enables predicting human-like fixes among the top few suggestions.
- Empirical evidence that the technique works well for a diverse set of bug categories taken from Error Prone and Infer.

To the best of our knowledge, Getafix is the first industrially-deployed automated bug-fixing tool that learns fix patterns from past, human-written commits, and produces human-like fixes in a short amount of time.

2 OVERVIEW

Getafix consists of three main components, organized in a learning phase and a prediction phase. In the following we will describe their functionality and challenges at a high level, followed by a more detailed description in later sections. Figure 2 gives an overview of the approach. During the learning phase, a set of pairs of bugs and their fixes is given as training data to Getafix. As training data can serve any collection of past human code changes tied to a specific signal, such as a static analysis warning, a type error, a lint message, or simply the fact that a change was suggested during human code review. Our evaluation focuses on static analysis warnings as the signal, i.e., all bugs and fixes have been detected as instances of a specific bug category by a static analyzer, e.g.,

[1] https://code.fb.com/developer-tools/open-sourcing-facebook-infer-identify-bugs-before-you-ship/

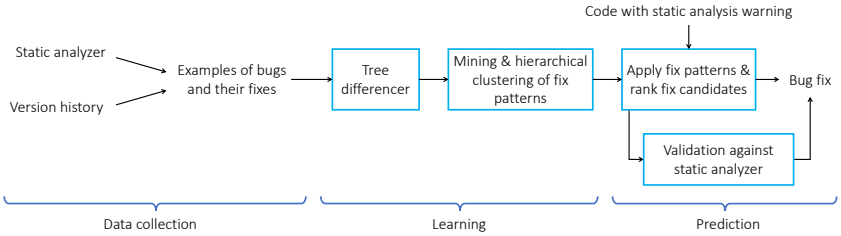


Fig. 2. Overview of Getafix.

as potential null dereferences. During the prediction phase, Getafix then takes previously unseen code that comes with the same signal as the training examples and produces a bug fix.

The first step of the learning phase, and the first of the three main Getafix components, is a *tree differencer*, which identifies changes at the AST level. It generates *concrete edits*, which are pairs of sub-ASTs of the original *before* and *after* ASTs, representing a specific change that could be replayed on different code by replacing instances of the edit's *before* AST with its *after* AST.

Based on the concrete, AST-level edits, the second component is a new way of *learning fix patterns*. To generalize from specific fix examples, fix patterns have "holes", i.e., pattern variables, that may match specific subtrees, similar to the representation used by Rolim et al. [2017] and Long et al. [2017]. A key technical contribution is a novel hierarchical, agglomerative clustering technique that organizes fix patterns into a hierarchical structure. The approach derives different variants of fix patterns, ranging from very specific patterns that match few concrete edits to very general fix patterns that match many concrete edits. Another key contribution is to include into the fix patterns not only the code changes themselves, but also some surrounding context (examples of which can be seen in Figure 3). This context is crucial to decide which out of multiple possible fix patterns to apply to a given piece of code.

After learning fix patterns, which is a once-per-bug-category effort, the last component of Getafix is a prediction phase which *applies the patterns* to previously unseen buggy code to produce a suitable fix. Since Getafix aims at predicting fixes without an expensive validation of many fix candidates, it internally *ranks candidate fixes*. We present a simple yet effective, statistical ranking technique that uses the additional context extracted along with fix patterns.

Before suggesting a fix to the developer, Getafix validates the predicted fix against the same tool that provided the signal for the bug category, e.g., a static analysis, type checker, or linter. Getafix is agnostic to the signal used, so we assume a possibly expensive black box component.

3 TREE DIFFERENCER

The first step of Getafix takes a set of example bug fixes, and decomposes each fix into fine-grained edits. These edits provide the basic ingredients for learning and applying fix patterns in later steps. For example, Figure 3 shows a code change and the three fine-grained edits that Getafix extracts: (i) inserting an early return if `task` is `null` (shown in green); (ii) changing `public` to `private` (shown in red); and (iii) moving the `doWork` method (shown in blue).

3.1 Trees

Getafix extracts fine-grained edits based on ASTs. For our purposes, a node in an AST has:

- a *label*, e.g., "BinEx" (for binary expression), "Literal", or "Name",
- a possibly empty *value*, e.g., `+`, `42`, or `foo`, and

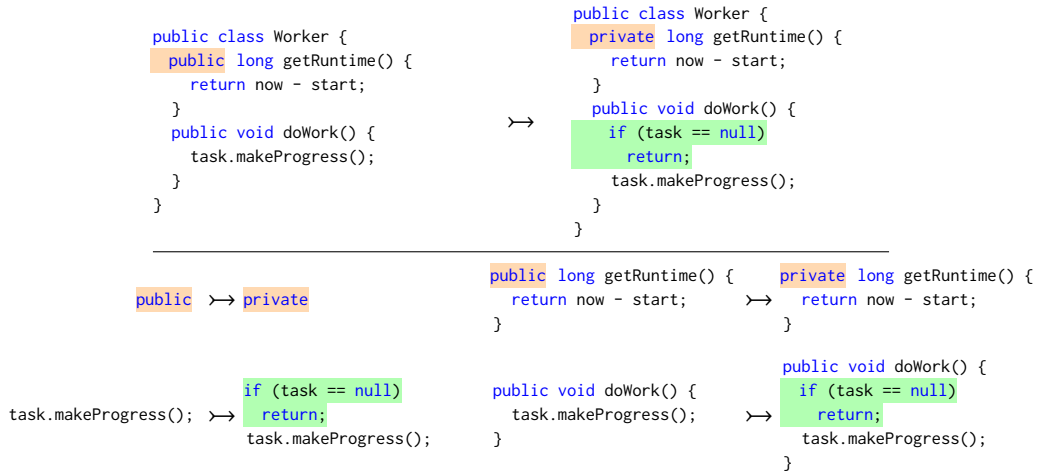


Fig. 3. Examples of concrete edits extracted by the tree differencer. The entire original file pair with modifications according to the tree differencer is at the top and concrete edits with varying levels of context (such as the fact that adding an empty empty return occurrs in a function with type void.)

- *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.

More formally, we define the set of trees as follows.

Definition 3.1 (Tree).

$$\text{Tree} = \text{Label} \times \text{Value} \times \bigcup_{k \in \mathbb{N}} (\text{Location} \times \text{Tree})^k$$

where Label, Location and Value are sets of strings.

For readability, we represent ASTs using term notation rather than tuples. For example, parsing `x = y + 2` results in an AST $\text{Assign}(\text{x} : \text{Name}, + : \text{BinEx}(\text{y} : \text{Name}, 2 : \text{Literal}))$. Child nodes are listed in parentheses following their parent node. Parentheses are omitted if no child nodes exist. Values, if not empty, are prepended to labels using syntax borrowed from type judgments.

3.2 Tree Edits

Given two trees, we define edits as follows.

Definition 3.2 (Tree edits).

$$\text{Edit} = \text{Tree} \times \text{Tree} \times \mathcal{P}^{\text{Mapping}}$$

$$\text{Mapping} = \text{TreeRef} \times \text{TreeRef} \times \{\text{mod}, \text{unmod}\}$$

Edits are triples containing the *before* and *after* ASTs, followed by a 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 subtrees is part of a modification (`mod` means the node is part a modification, `unmod` means that the mapped subtree is unmodified). TreeRef is the set of references uniquely identifying a node within an AST.

We write edits in source code as `before code` ⟼ `after code`, e.g., `x = y + 2` ⟼ `x = 3 + y`.

Combining this notation with our AST notation from above, we write tree edits as `before AST` ⟼ `after AST`, where we write modified subtrees bold (whenever the distinction is relevant) and give

nodes connected by a mapping matching indices. For example, two concrete edits emitted for the above code change are $\textbf{+ : BinEx}_0(\textbf{y : Name}_1, \textbf{2 : Literal}) \rightarrowtail \textbf{+ : BinEx}_0(\textbf{3 : Literal}, \textbf{y : Name}_1)$ and
$\textbf{Assign}_0(\mathsf{x} : \mathsf{Name}_1, \textbf{+ : BinEx}_2(\textbf{y : Name}_3, \textbf{2 : Literal}))$
$\rightarrowtail \textbf{Assign}_0(\mathsf{x} : \mathsf{Name}_1, \textbf{+ : BinEx}_2(\textbf{3 : Literal}, \textbf{y : Name}_3))$.

An alternative to tree-based reasoning about edits would be to use a line-based diffing tool. However, line-based diffing reasons about code on a more coarse-grained level, ignoring the structural information provided by ASTs. For example, given the change in Figure 3, line-based diffing would mark both methods as fully removed and inserted, whereas tree-based edits can encode the move and hence also detect the insertion within the moved method as a concrete edit.

3.3 Extracting Concrete Edits

To extract edits from a given pair of ASTs, Getafix builds on the GumTree algorithm [Falleri et al. 2014], a tree-based technique to compute fine-grained edit steps that lead from one AST to another. The approach extracts four kinds of edit steps: deletion, insertion, move, and update. 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 mapped to each other is considered a *move*. The pair is also considered a move if their parents are mapped, but the nodes have different subtree position in those parents (think for instance of a parameter swap). A mapped pair of nodes with different values is considered an *update* and may be part of a move at the same time. Any pairs of nodes involved in one of the above operations is consider to be *modified*, whereas all other pairs of mapped nodes are considered *unmodified*. An entire subtree is considered *unmodified* if all its nodes are unmodified.

For the example in Figure 3, the insertion of the if statement is an addition, the change from `public` to `private` is an update, and method `doWork` has been moved. The subtree representing the call `task.makeProgress()` is unmodified. In contrast, the subtree representing the block surrounding this call is modified, due to the insertion of the if statement.

As demonstrated with the insertion combined with a move in Figure 3, modifications can be nested and it is unclear what the granularity of a concrete edit should be. Maybe the insertion itself is the fix, maybe the move is important. Our algorithms are language agnostic, so without domain knowledge there is no robust strategy for grouping modifications into concrete edits. We address this challenge by extracting an entire *spectrum* of concrete edits based on the modifications reported by GumTree: Any pair of mapped nodes will be turned into the roots of a new concrete edit, if that concrete edit contains at least one modification. In the example of Figure 3, we would hence extract concrete edits rooted at the body of `doWork` (contains the insertion), at the method declaration of `doWork` (contains the insertion as well), at the method declaration of `getRuntime` (contains the update), at the class body level (contains all modifications), and any of its ancestors for the same reason. By contrast, we will not emit a concrete edit rooted at the body of `getRuntime` as it does not contain a modification. In addition to making concrete edits rooted at mapped nodes, we also create a concrete edit rooted at the parent of each connected component of modified nodes in the before and after trees. This is done to ensure that we still learn patterns from edits which occur near other changes.

The approach is designed to extract too many rather than too few concrete edits. The rationale is that the clustering step of Getafix, which is explained in detail in the following, automatically prioritizes patterns with the best level of granularity, since there will exist multiple similar concrete edits. For instance, when combining the concrete edits emitted for Figure 3 with further concrete edits from bug fixes for null dereferences, the insertion of an if statement is likely to appear more often than the move or update. In contrast, concrete edits containing further modifications, e.g.,

rooted further up in the AST, are likely to be discarded as noise, because there are no similar concrete edits to create a cluster with.

4 LEARNING FIX PATTERNS

Given the set of fine-grained, tree-level edits extracted by the tree differencer described in the previous section, the next step is to learn fix patterns, i.e., recurring edit patterns observed in fixes for a specific kind of bug. Intuitively, an edit pattern can be thought of as a generalization of multiple edits. To abstract away details of concrete edits, edit patterns may have "holes", or pattern variables, to represent parts of a tree where concrete edits differ. A key contribution of Getafix is a novel algorithm that derives a hierarchy of edit patterns, which contains edit patterns at varying levels of generality, ranging from concrete edits at the leaves to abstract edit patterns at the root.

The algorithm to derive a hierarchy of edit patterns is based on a generalization operation on edit patterns, which we obtain through anti-unification [Kutsia et al. 2014], an existing method of generalizing among different symbolic expressions. We start by presenting the generalization operation (Section 4.1) and then show how Getafix uses this operation to guide a hierarchical clustering algorithm (Section 4.2). Finally, Section 4.3 describes how Getafix augments edit patterns with context information that helps deciding when to apply a learned edit pattern to new code.

4.1 Generalizing Edit Patterns via Anti-Unification

4.1.1 Tree Patterns and Tree Edit Patterns. A set of concrete trees can be abstracted into a tree pattern. Formally, we extend our definition of trees by adding holes:

Definition 4.1 (Tree pattern). Let $\mathsf{Hole} = (\mathsf{Label} \cup \{?\}) \times \mathbb{N}$, then the set of tree patterns is

$$\mathsf{Tree}^+ = \mathsf{Label} \times \mathsf{Value} \times \bigcup_{k \in \mathbb{N}} (\mathsf{Location} \times \mathsf{Tree}^+)^k \cup \mathsf{Hole}$$

Holes may have a label α (we write $h^k : \alpha$) in which case they match an arbitrary subtree that has a root with label α. If the label is omitted (we write $h^k : ?$) the hole matches any subtree. Holes are indexed, allowing tree patterns to express whether two holes must match identical subtrees.

For example, consider the following trees $t_1, t_2, t_3 \in \mathsf{Tree}$ and tree patterns $p_1, p_2, p_3, p_4 \in \mathsf{Tree}^+$:

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

Pattern p_1 matches only t_1, since t_2 and t_3 do not match the label of hole h^1. Patterns p_2 and p_3 match all three ASTs. Finally, p_4 matches only t_3, since the pattern requires both operands of the binary expression to be identical.

Similar to tree patterns, we define the set Edit^+ of edit patterns analogous to Edit (see Definition 3.2), but using Tree^+ instead of Tree.

4.1.2 Generalizing Tree Patterns. To learn recurring edit patterns, Getafix generalizes concrete edits into patterns through a process called anti-unification. As an intermediate step, we first present how to generalize two tree patterns into a generalization that describes both of them. The

$$
\begin{aligned}
&\text{antiUnify}(\ \boldsymbol{v : \alpha(a_1, \ldots, a_n)}\ ,\ \boldsymbol{w : \beta(b_1, \ldots, b_m)}\ ) = \\
&\quad \begin{cases}
\boldsymbol{v : \alpha(c_1, \ldots, c_n)} & \text{if } \alpha = \beta \wedge v = w \wedge n = m \\
\qquad \text{where } c_i = \text{antiUnify}(a_i, b_i)\ \ \forall\ i \in \{1, \ldots, n\} & \\
\boldsymbol{h^k : \alpha} & \text{if } \alpha = \beta \wedge (v \neq w \vee n \neq m) \\
\boldsymbol{h^k : ?} & \text{otherwise}
\end{cases} \\
&\qquad \text{where } n, m \in \mathbb{N},\quad a_1, \ldots, a_n, b_1, \ldots, b_m \in \text{Tree}^+ \\
&\qquad\qquad \alpha, \beta \in \text{Label},\quad v, w \in \text{Value},\quad \text{fresh hole index } k \in \mathbb{N} \\
&\text{antiUnify}(\ \boldsymbol{v : \alpha(a_1, \ldots, a_n)}\ ,\ \boldsymbol{h^t : \beta}\ ) = \\
&\text{antiUnify}(\ \boldsymbol{h^s : \alpha}\ ,\ \boldsymbol{w : \beta(b_1, \ldots, b_m)}\ ) = \\
&\text{antiUnify}(\ \boldsymbol{h^s : \alpha}\ ,\ \boldsymbol{h^t : \beta}\ ) = \\
&\quad \begin{cases}
\boldsymbol{h^k : \alpha} & \text{if } \alpha = \beta \\
\boldsymbol{h^k : ?} & \text{otherwise}
\end{cases} \\
&\qquad \text{where } n, m, s, t \in \mathbb{N},\quad a_1, \ldots, a_n, b_1, \ldots, b_m \in \text{Tree}^+ \\
&\qquad\qquad \alpha, \beta \in \text{Label},\quad v, w \in \text{Value},\quad \text{fresh hole index } k \in \mathbb{N}
\end{aligned}
$$

Fig. 4. Anti-unification of tree patterns. For conciseness, we assume the antiUnify function performs memoization, so that it reuses holes where possible, without explicitly tracking them.

generalization is constructed such that each hole has a set of substitutions that can be used to recover the original input tree patterns. There always exists a trivial generalization, which has 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.

Figure 4 formally describes the anti-unification function for tree patterns. The function is defined recursively on the tree structure. It merges subtrees that have the same labels, the same values, and the same number of children into a combined subtree, and replaces them by a hole otherwise. Merging a subtree with a subtree represented by a hole yields another hole. When merging two holes, the function keeps the label of the hole whenever it matches.

For example, let

$$a = \text{Assign}(\text{a} : \text{Name}, + : \text{BinEx}(\text{a} : \text{Name}, \text{a} : \text{Name}))$$

and

$$b = \text{Assign}(\text{b} : \text{Name}, + : \text{BinEx}(\text{b} : \text{Name}, 2 : \text{Literal}))$$

be the ASTs for assignment statements `a = a + a;` and `b = b + 2;`, respectively. Then

$$\text{antiUnify}(a, b) = \text{Assign}\big(h^0 : \text{Name}, + : \text{BinEx}\big(h^0 : \text{Name}, h^1 : ?\big)\big)$$

Note how hole $h^0 : \text{Name}$ can be reused since in both instances it substitutes variables `a` and `b`. While $h^1 : ?$ also substitutes `a` in AST a, it does not substitute `b` in AST b, so a fresh hole is created. Because the hole's substitutions have mismatching labels, `?` is used.

Instead of the above pattern, $\text{Assign}\big(h^0 : \text{Name}, + : \text{BinEx}\big(h^1 : \text{Name}, h^2 : ?\big)\big)$ also generalizes a and b, but the information that $h^0 : \text{Name}$ and $h^1 : \text{Name}$ substitute identical subtrees would be lost. Pattern $\text{Assign}\big(h^0 : ?, + : \text{BinEx}\big(h^1 : ?, h^2 : ?\big)\big)$ would be an even more general generalization,

$$\text{antiUnify}(\mathit{before}_1 \rightarrowtail \mathit{after}_1, \mathit{before}_2 \rightarrowtail \mathit{after}_2) =$$
$$\text{populate}(\mathit{before}_g \rightarrowtail \mathit{after}_g, \mathit{before}_1 \rightarrowtail \mathit{after}_1, \mathit{before}_2 \rightarrowtail \mathit{after}_2)$$

where

$$\mathit{before}_g = \text{antiUnify}(\text{stripUnmod}(\mathit{before}_1), \text{stripUnmod}(\mathit{before}_2))$$
$$\mathit{after}_g = \text{antiUnify}(\text{stripUnmod}(\mathit{after}_1), \text{stripUnmod}(\mathit{after}_2))$$
$$\text{for } \mathit{before}_1 \rightarrowtail \mathit{after}_1, \mathit{before}_2 \rightarrowtail \mathit{after}_2 \in \text{Edit}^+$$

$$\text{stripUnmod}(\boldsymbol{v} : \boldsymbol{\alpha}(\overline{\boldsymbol{a_{1..i-1}}}, a_i, \overline{\boldsymbol{a_{i+1..n}}})) = \text{stripUnmod}(\boldsymbol{v} : \boldsymbol{\alpha}(\overline{\boldsymbol{a_{1..i-1}}}, \overline{\boldsymbol{a_{i+1..n}}}))$$

(drop unmodified children)

$$\text{stripUnmod}(\boldsymbol{v} : \boldsymbol{\alpha}(\boldsymbol{a_1}, \ldots, \boldsymbol{a_n})) = \boldsymbol{v} : \boldsymbol{\alpha}(\textbf{stripUnmod}(\boldsymbol{a_1}), \ldots, \textbf{stripUnmod}(\boldsymbol{a_n}))$$

(if no more children are unmodified, recurse)

$$\text{for } n, i \in \mathbb{N}, \quad a_1, \ldots, a_n \in \text{Tree}^+, \quad \alpha \in \text{Label}, \quad v \in \text{Value}$$

$$\text{populate}(e_g, e_1, e_2) = \text{Add back unmodified nodes and mappings}$$

Fig. 5. Anti-unification of edit patterns. To reuse holes created for the *before* and *after* trees, we assume that antiUnify uses memoization across calls to the function.

dropping the information about labels even though they matched. Instead of computing these generalizations, anti-unification produces the least general generalization of the given patterns.

4.1.3 Generalizing Edit Patterns. We now extend the idea of generalizing tree patterns to tree edit patterns, which yields the generalization operation at the core of Getafix's learning algorithm. Figure 5 formally defines anti-unification of edit patterns. The basic idea is to first anti-unify the *before* trees and then the *after* trees, while using a single set of substitutions between the *before* and *after* trees, to indicate that a hole in the *before* tree corresponds to the same AST node in the *after* tree. To focus the generalized edit patterns on the modified parts of a tree, the approach uses a helper function stripUnmod to drop unmodified subtrees before anti-unifying the trees. Afterwards, Getafix populates the unmodified nodes back and maps them to each other.

As an example, assume the training data contains two source edits:
`{ f(); g(); x = 1; }` ↣ `{ f(); x = 1; if (c) g(); }`
and
`{ return; y = 2; }` ↣ `{ y = 2; if (c) onResult(); }`. The corresponding concrete edits are the following:

$$\mathit{before}_1 = \textbf{Block}_0(\text{f} : \text{Call}_1, \textbf{g : Call}_2, \text{Assign}_3(\text{x} : \text{Name}_4, 1 : \text{Num}_5))$$
$$\mathit{after}_1 = \textbf{Block}_0(\text{f} : \text{Call}_1, \text{Assign}_3(\text{x} : \text{Name}_4, 1 : \text{Num}_5), \textbf{If}(\textbf{c : Name}, \textbf{g : Call}_2))$$
$$\mathit{before}_2 = \textbf{Block}_6(\textbf{Return}, \text{Assign}_7(\text{y} : \text{Name}_8, 2 : \text{Num}_9))$$
$$\mathit{after}_2 = \textbf{Block}_6(\text{Assign}_7(\text{y} : \text{Name}_8, 2 : \text{Num}_9), \textbf{If}(\textbf{c : Name}, \textbf{onResult : Call}))$$

We first drop the unmodified subtrees and anti-unify the result:

$$\text{stripUnmod}(\mathit{before}_1) = \mathbf{Block}_0(\mathbf{g : Call}_2)$$
$$\text{stripUnmod}(\mathit{after}_1) = \mathbf{Block}_0(\mathbf{If}(\mathtt{c} : \mathbf{Name}, \mathbf{g : Call}_2))$$
$$\text{stripUnmod}(\mathit{before}_2) = \mathbf{Block}_6(\mathbf{Return})$$
$$\text{stripUnmod}(\mathit{after}_2) = \mathbf{Block}_6(\mathbf{If}(\mathtt{c} : \mathbf{Name}, \mathtt{onResult} : \mathbf{Call}))$$
$$\mathit{before}_g = \text{antiUnify}(\text{stripUnmod}(\mathit{before}_1), \text{stripUnmod}(\mathit{before}_2)) = \mathbf{Block}\big(\boldsymbol{h}^0 : ?\big)$$
$$\mathit{after}_g = \text{antiUnify}(\text{stripUnmod}(\mathit{after}_1), \text{stripUnmod}(\mathit{after}_2)) = \mathbf{Block}\big(\mathbf{If}\big(\mathtt{c} : \mathbf{Name}, \boldsymbol{h}^1 : \mathbf{Call}\big)\big)$$

We then attempt to populate back mappings where possible. Mappings are re-established between any pair of nodes that are the result of anti-unifying mapped nodes. Here, both edits agree about the blocks mapping to each other (mappings 0 and 6):

$$\mathit{before}'_g = \mathbf{Block}_{10}\big(\boldsymbol{h}^0 : ?\big)$$
$$\mathit{after}'_g = \mathbf{Block}_{10}\big(\mathbf{If}\big(\mathtt{c} : \mathbf{Name}, \boldsymbol{h}^1 : \mathbf{Call}\big)\big)$$

Finally, Getafix populates back unmodified nodes where possible. Both edits have an assignment statement, , : $\text{Assign}_{11}(t)$ hat swaps places with the modified statement. The approach anti-unifies this unmodified node and adds it back, giving the final edit pattern $\mathit{before}''_g \rightarrowtail \mathit{after}''_g$:

$$\mathit{before}''_g = \mathbf{Block}_{10}(\boldsymbol{h}^0 : ?, \text{Assign}_{11}\big(h^2 : \text{Name}, h^3 : \text{Num}\big))$$
$$\mathit{after}''_g = \mathbf{Block}_{10}(\text{Assign}_{11}\big(h^2 : \text{Name}, h^3 : \text{Num}\big), \mathbf{If}\big(\mathtt{c} : \mathbf{Name}, \boldsymbol{h}^1 : \mathbf{Call}\big))$$

4.2 Hierarchical Clustering of Edit Patterns

Based on the generalization operation described above, Getafix uses a hierarchical clustering algorithm to derive a hierarchy of recurring edit patterns. The basic idea is to start with all concrete edits and to iteratively combine pairs of edits until all edits are combined into a single edit pattern. At each iteration, the algorithm picks a pair of edit patterns to combine, so that their anti-unification yields the least loss of concrete information. The sequence of generalization steps eventually yields a hierarchy of edit patterns, where leaf nodes represent concrete edits and higher-level nodes represent increasingly generalized edit patterns.

Before delving into our algorithm, consider the example hierarchy in Figure 6, which visualizes a learned hierarchy of edit patterns as a so-called dendrogram. Each pair of red and green boxes represents one edit pattern. The dendrogram is derived from four concrete edits, which are shown as edit patterns without any holes in them. The left-most edit pattern is the most general description of all four concrete edits, while all edit patterns in between represent different levels of generalization that comprise different subsets of the concrete edits. The main benefit of keeping intermediate edit patterns, instead of greedily merging all edits into a single generalization, is that Getafix can choose the most suitable pattern to apply to a new piece of code (as explained in detail in Section 5).

4.2.1 Clustering Algorithm. To obtain a hierarchy of edit patterns from a given set of concrete edits, Getafix performs a hierarchical, agglomerative clustering algorithm. The algorithm maintains a working set W of edit patterns. Initially, this working set contains one edit pattern for each of the given concrete edits. The main loop of the algorithm repeats the following steps until W is reduced to a singleton set:

(1) Pick a pair e_1, e_2 of edit patterns from W.
(2) Generalize the edit patterns using anti-unification, which yields $e_3 = \text{antiUnify}(e_1, e_2)$.
(3) Remove e_1 and e_2 from W, and add e_3 to W instead.

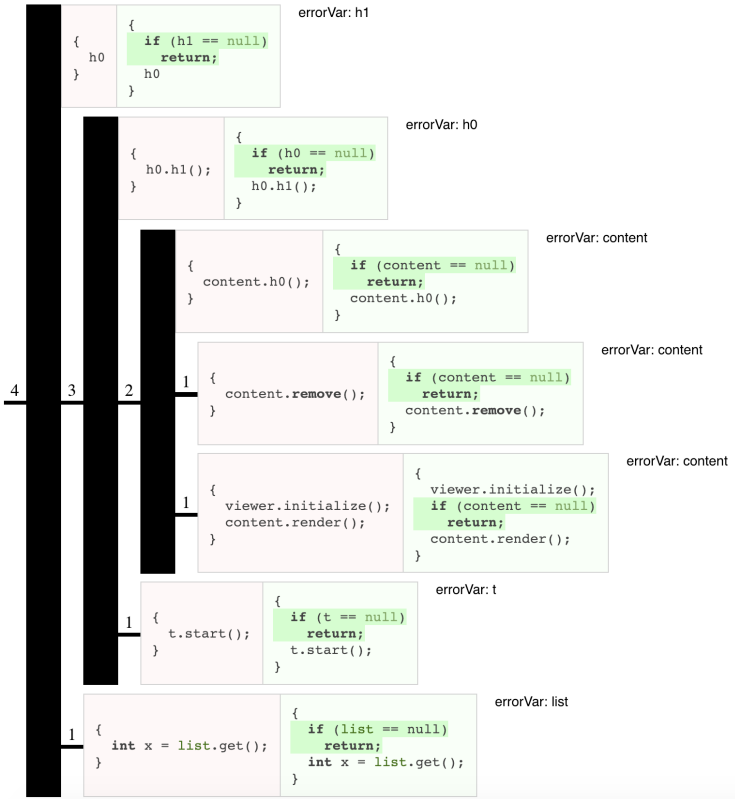


Fig. 6. Dendrogram showing concrete edits merged into more abstract edit patterns. The vertical black bars correspond to levels in the hierarchy, and the edit pattern at the top of each black bar is the pattern obtained by anti-unification of the edit pattern connected by the smaller, thin black lines.

(4) Set e_3 as the parent of e_1 and e_2 in the hierarchy.

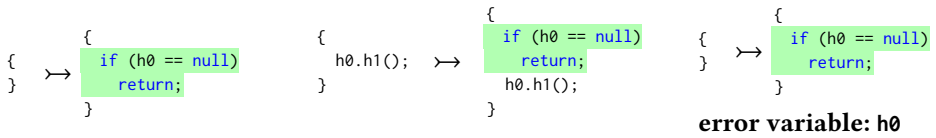
4.2.2 Merge Order. A crucial component of our clustering algorithm is how to pick the pair e_1, e_2 of edit patterns to generalize. Getafix aims at minimizing the loss of concrete information at each anti-unification step. To this end, we exploit the fact that the anti-unification operation itself gives a partial order of edit patterns, which orders patterns by their level of abstraction. Given this partial order of edit patterns, the algorithm picks a pair e_1, e_2 that produces a minimal generalization w.r.t. the partial order. In the following, we first describe the partial order of edit patterns and then present an efficient calculation of it, which our Getafix implementation is based on.

The partial order of edit patterns is based on a partial order of tree patterns, which builds upon the anti-unification operation:

Definition 4.2 (Partial order of tree patterns). Anti-unification represents a least upper bound operation on Tree^+. The resulting semilattice $\langle \mathsf{Tree}^+, \mathsf{antiUnify} \rangle$ has Tree as minimal elements and $h^k : ?$ (regardless of k) as its greatest element. $\langle \mathsf{Tree}^+, \mathsf{antiUnify} \rangle$ hence induces a partial order $\sqsubseteq$ on Tree^+ (read "more precise than") by defining $p \sqsubseteq p' \iff \mathsf{antiUnify}(p, p') = p'$.

The definition relies on equality of tree patterns. We consider two tree patterns $p_1, p_2 \in \mathsf{Tree}^+$ equal if they are structurally identical, modulo bijective substitution of hole indices. For example,

$$+ : \mathsf{BinEx}\big(h^0 : \mathsf{Name}, h^1 : ?\big) = + : \mathsf{BinEx}\big(h^5 : \mathsf{Name}, h^3 : ?\big)$$



(a) The hole h0 is unbound. (b) Code context binds h0. (c) Error context binds h0.

Fig. 7. Edit pattern with an unbound hole and two ways of creating a bound version by adding context.

$$+ : \mathsf{BinEx}\big(h^0 : \mathsf{Name}, h^1 : ?\big) = + : \mathsf{BinEx}\big(h^1 : \mathsf{Name}, h^0 : ?\big)$$

$$+ : \mathsf{BinEx}\big(h^0 : \mathsf{Name}, h^1 : ?\big) \neq + : \mathsf{BinEx}\big(h^2 : \mathsf{Name}, h^2 : ?\big)$$

As a consequence, all single-hole patterns $h^k : ?$ are identical.

Finally, we define the partial order of edit patterns analogous to Definition 4.2:

Definition 4.3 (Partial order of edit patterns). $\langle \mathsf{Edit}^+, \mathsf{antiUnify} \rangle$ induces a partial order $\sqsubseteq$ on Edit^+ (read "more precise than") by defining $e \sqsubseteq e' \iff \mathsf{antiUnify}(e, e') = e'$.

To compute this partial order quicky, Getafix uses the following steps, each acting as a tiebreaker for the previous step. When picking edit pairs to merge, Getafix prefers generalizations that

(1) yield patterns without any unbound holes in the *after* part over those with such holes,
(2) preserve more mappings between modified nodes,
(3) have fewer holes and holes generalizing away smaller subtrees,
(4) preserve more mappings between unmodified nodes with labels,
(5) preserve more error context (explained in Section 4.3), and
(6) preserve more mappings between unmodified nodes.

4.2.3 Approximations for Efficiency. To ensure that Getafix scales to a large number of edits, our implementation performs two approximations.

First, Getafix exploits the fact that after performing one merge, the new set of edit patterns and hence also the new set of merge candidates is largely unchanged. To reduce the number of times that the same pair of potential merges is compared, the implementation uses the nearest-neighbor-chain algorithm [Benzécri 1982]. This reduces the runtime complexity from cubic to quadratic.

Second, because the clustering algorithm is quadratic in the number of concrete edits, we reduce the number of edits considered at once. Specifically, the implementation partitions all given edits by the labels of their modified nodes in the *before* and *after* trees. As edits for which the labels differ get least priority in the merge order, Getafix assigns them to separate partitions and constructs a hierarchy for each partition separately. Finally, the approach generalizes the roots of these hierarchies to form a single dendrogram. This second approximation may appear trivial and not something that can impact results, but according to our partial order a merge of trees with different labels could be preferred over a merge of two trees with the same labels. Consider the three *before* trees t1: `a.b.c();` t2: `d.e.f();` t3: `g.h.b.c();`. The possible merge results are as follows:

- t1 + t2: `h0.h1.h2();`
- t1 + t3: `h0.b.c();`
- t2 + t3: `h0.h1.h2();`

In this case we will prefer the merge of t1 + t3 since it introduces least number of holes, even though t1 and t3 have different labels and t1 and t2 have the same labels.

4.3 Additional Context in Edit Patterns

In addition to a bare description of modified code, the edit patterns learned by Getafix also include context information. The motivation is twofold: First, context makes edit patterns more specific, and hence, they match fewer code locations, which reduces the chance to produce invalid fixes. Second, context information can bind otherwise unbound holes, which helps applying a pattern to new code. Getafix uses two kinds of context: *code context* and *error context*.

Code context. Because Getafix starts with an entire spectrum of concrete edits, which include different amounts of code in addition to the changed AST nodes, the learned edit patterns also include contextual code. Such contextual code helps deciding when a pattern is applicable, and it helps applying the pattern to new code. Figure 7 gives an example of an edit pattern with an unbound hole that becomes bound thanks to additional code context. In edit pattern (a), `h0` is unbound in the *after* part, leaving it unclear what to insert as `h0` when applying the pattern. In contrast, edit pattern (b) binds `h0` by including unmodified code in the *before* part.

Error context. The static analysis warning addressed by Getafix may also provide additional context information. For example, the expression blamed for a `NullPointerException` gives a hint about how and where to apply a fix pattern. Getafix optionally considers such hints in the form of an error variable. While learning edit patterns, the approach then propagates this information, eventually obtaining patterns where a specific hole represents the error variable. Figure 7 gives an example for such error context. Pattern (c) specifies that `h0` is the error variable. As a result, Getafix can apply the pattern even though `h0` is not mentioned in the *before* part.

4.4 Comparison with Prior Work on Inferring Edit Patterns

An alternative to our hierarchical clustering is a greedy clustering algorithm, as suggested for inferring recurring edit patterns [Rolim et al. 2018]. That approach maintains a single representation of each cluster, and this representation does not contain any context information. Even if one would add context information, the approach could add only context that is present in all of the edits in the training data, which is unlikely to learn edit patterns with the same level of context as Getafix.

For example, in Figure 6, the edit at the top of the hierarchy simply says to insert a check for null before some statement `h0`. This is the only pattern that would be learned if we had only a single representation of this cluster. Instead, hierarchical clustering enables Getafix to observe the pattern one level down, where the early return is inserted before a method call `h0.h1()`, which uses the variable that could be null. Such extra context is crucial for predicting a human-like fix. Section 6.4 compares our approach with greedy clustering.

5 APPLYING AND RANKING FIX PATTERNS

Given buggy source code and a set of learned fix patterns, the final step of Getafix produces a fixed version of the source code. The main challenges of this step are (i) finding which patterns are applicable to a given piece of code (Section 5.1) and (ii) deciding which of all fix candidates produced by applying these patterns to suggest to the developer (Section 5.2).

5.1 Applying Edit Patterns to Buggy Code

To find applicable edit patterns, Getafix matches tree patterns to the tree representation of buggy source code.

Definition 5.1 (Matching). Let $p \in \mathsf{Tree}^+$ and $t \in \mathsf{Tree}$. Then p matches t if there exists a function $subst : \mathsf{Hole} \longrightarrow \mathsf{Tree}$ so that replacing every hole h in p with $subst(h)$ yields t. Intuitively, $subst$ captures the substitutions one has to perform to concretize p into t.

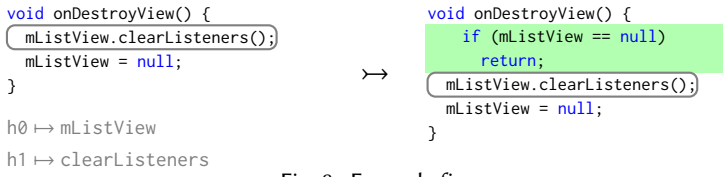


Fig. 8. Example fix.

Given a specific edit pattern p and a buggy tree t, the approach creates a fix using the following steps:

(1) Find a subtree t_{sub} in t that matches the *before* part of p.
(2) Consistently instantiate all holes in the *before* part and the *after* part of p
(3) Replace the subtree t_{sub} with the instantiated *after* part.

As an example, consider the code on the left of Figure 8 and one of the edit patterns mentioned earlier:[2]

```
h0.h1(); ↣ if (h0 == null) return; h0.h1();
```

The *before* part of the edit pattern matches the buggy code by substituting `h0` with `mListView` and `h1` with `clearListeners`. Instantiating the *after* part of the pattern with these substitutions and replacing the subtree that matches the *before* part with the instantiated *after* part results in the code on the right of Figure 8.

5.2 Ranking Fix Candidates

By applying edit patterns, Getafix obtains a set F_{cand} of applicable fixes, which we call fix candidates. In practice, multiple patterns in a learned hierarchy of edit patterns may apply to a given piece of buggy code. In fact, one of the main benefits of learning multiple edit patterns per bug category is to enable Getafix to select the most suitable pattern for a given situation. However, having multiple applicable patterns leads to the challenge of deciding which fix to suggest to a developer.

To illustrate the problem, consider multiple learned fix patterns for null dereferences:

p_1: `(empty) ↣ if (h0==null) { return; }` where `h0` is the error variable
p_2: `h0.h1() ↣ h0!=null && h0.h1()`
p_3: `h0.h1(); ↣ if (h0==null) { return; } h0.h1();`

Some patterns match more often than others. Patterns without unmodified nodes, such as p_1, match often and may even match multiple times within a given buggy piece of code. For the code on the left of Figure 8, p_1 matches in unintended places, such as after `mListView.clearListeners()` or even after `mListView = null`. In contrast, p_2 and p_3 are more specific, because they add the null check relative to existing code fragments. Choosing which of these patterns is the most appropriate is a non-trivial problem.

We address the challenge of ranking fix candidates through a simple yet effective ranking algorithm that sorts fix candidates by their likelihood to be relevant, based on the human-written fixes that Getafix learns from. Given a bug b and a set P of fix patterns, the problem is to rank all fix candidates F_{cand} that result from applying some $p \in P$ to the code that contains b. Let a tuple (b, p, z) represent the application of pattern p to the code that contains b at source code location z. The location z is relative to the location where the static analyzer reports a warning about bug b. For example, a location $z = -1$ means that the fix pattern is applied one line above the warning location, whereas $z = 3$ means the fix pattern is applied three lines below the warning. The goal of

[2]To simplify the presentation, we will from now on represent edit patterns as pseudo-code instead of term notation.

the ranking is that the probability $\mathbb{P}(\mathsf{hfix}(b,p,z) \mid \mathsf{match}(b,p,z))$ is higher for fix candidates (b,p,z) that appear higher up in the ranking, where match indicates that the pattern is applicable and hfix indicates that the fix is the same a human developer would implement. Obviously, precisely computing hfix is impossible in general, because it depends on a human decision. Instead, Getafix estimates the likelihood that a fix candidate will be accepted by a human based on the set H of human fixes that the approach learns from. To estimate how likely a fix candidate matches a human fix, Getafix computes three scores:

- *Prevalence score.* This score is higher for patterns that cover more concrete bug fixes in the set H of human training fixes:

$$s_{prevalence} = \frac{|\{\text{instances of p in } H\}|}{|H|}$$

  The intuition is that a more common way of fixing a specific kind of bug is more likely to appeal to developers.
- *Location score.* This score measures what fraction of bugs that were fixed with pattern p were fixed by applying the pattern exactly z lines away from the warning location:

$$s_{location} = \frac{|\{\text{instances of } p \text{ at location } z \text{ in } H\}|}{|\{\text{instances of } p \text{ in } H\}|}$$

  The motivation for this score is that some fix patterns typically occur only at specific locations. For example, fixing a null dereference by inserting an early return often occurs just above the location where the static analyzer warns about the potential null dereference. We describe below how Getafix computes a distribution of line distance for each learned pattern.
- *Specialization score.* This score is higher for more specialized patterns, i.e., patterns that are applicable at fewer code locations.

$$s_{specialized} = \frac{|\{\text{all AST nodes}\}|}{|\left\{\begin{array}{c}\text{AST subtrees that match}\\ \text{the } \textit{before} \text{ part of } p\end{array}\right\}|}$$

  The intuition is that a more specific pattern contains more context information, and that if such a pattern matches, it is more likely to fit the given bug.

We estimate $\mathbb{P}(\mathsf{hfix}(b,p,z) \mid \mathsf{match}(b,p,z)) \propto s_{prevalence} * s_{location} * s_{specialized}$, so the product of these scores is used for ranking fix candidates, with higher scores being ranked higher. The location score is based on the distribution of locations, relative to the warning location, where a fix pattern is typically applied. Getafix knows this location for each concrete fix in the training data, and then propagates the locations while generalizing fixes into fix patterns. Specifically, we model the location as three statistical values tracked for each pattern during pattern learning. First, we track the ratio of fixes applied before and fixes applied after the warning location. This ratio is useful because some patterns typically apply either before or after the warning. For concrete fixes, we initialize this ratio to 0, 1, and 0.5 if the human fix is above, below, or on the same line as the warning, respectively. Second and third, we track two geometric distributions that model the line distance above and below the warning location. For concrete fixes, we initialize these distributions as having their expected value exactly at the distance where the concrete fix is applied. Whenever the clustering algorithm (Section 4.2) merges two edit patterns, it statistically propagates the three values from the two merged patterns to the resulting pattern.

As an example for the line distance distributions, reconsider fix pattern p_1 from above, which addresses a potential null deference by inserting an if statement that returns from the method if `h0` is null. Because this pattern is typically applied before the warning location, the above/below

ratio will be close to zero and the distribution for the line distance above will contain mostly small, positive integers.

Example 5.2 (Ranking among candidates).
Assume that we are given the code from Figure 8 to fix and know the three fix patterns $p1, p2$ and $p3$ we motivated ranking with. Applying $p1$ results in three fix candidates $p1_1, p1_2, p1_3$ as mentioned earlier: The "early return" can be inserted above, between and below both statements. $p2$ and $p3$ only match on the buggy line, resulting in fix candidates $p2_1$ and $p3_1$.

- For $p1_1$ we assume the following scores:

  $$s_{prevalence} = 0.03 \quad s_{location} = 0.5 \quad s_{specialized} = 20$$

  This would mean that we observed $p1$ being used 3% of the time. Of those times, the statement was inserted right above the blamed line 50% of the time. Furthermore the pattern matches one in twenty AST nodes.
- For $p1_2$ and $p1_3$ only $s_{location}$ changes since the other scores exclusively depend on the pattern used:

  $$s_{prevalence} = 0.03 \quad s_{location} = 0 \quad s_{specialized} = 20$$

  Since insertion of an "early return" was never observed *below* the buggy line, $s_{location}$ is zero.
- We assume $p2$ was observed 5% of the time, almost always being applied to exactly the buggy line and $p2$ matches one in forty AST nodes on average. Resulting scores for $p2_1$:

  $$s_{prevalence} = 0.05 \quad s_{location} = 0.95 \quad s_{specialized} = 40$$

- As $p3$ is more specialized than $p1$, we assume it was used for 2% of fixes, but also matches only one in 200 AST nodes. The specialization contains the buggy method call, so it was also applied right there 90% of the time. Resulting scores for $p3_1$:

  $$s_{prevalence} = 0.02 \quad s_{location} = 0.9 \quad s_{specialized} = 200$$

We conclude that Getafix would rank candidates in order $p3_1, p2_1, p1_1$ and so on due to respective overall scores 3.6, 1.9, 0.3 and 0.

5.3 Comparison with Other Ranking Techniques

The ranking step of Getafix relates to prior work on ranking fix candidates in the context of automated program repair. In principle, other ways of ranking may be plugged into Getafix, either to replace and to complement our current approach. One candidate for such an enhancement is the Prophet ranking [Long and Rinard 2016], which uses a set of over 3000 features of code and code changes to predict which fixes are more likely to be adequate. We did not add Prophet to our current prototype because the existing Getafix ranking works well in practice (Section 6.3) and because adapting Prophet's over 3000 features to the Java programming language is non-trivial. Conceptually, there are two advantages of our ranking over Prophet. First, Getafix ranks candidate fixes based on human fixes for the same kind of bug, whereas Prophet relies on a generic model of how "natural" a code change is. Learning to rank from fixes for the same bug category allows Getafix to specialize to bug category-specific "features". Second, our ranking is defined based on ASTs and line numbers, i.e., in a language-agnostic way, making it easy to apply the approach to another language.

6 EVALUATION

We address the following research questions:

- **RQ1**: How effective is Getafix at predicting human-like fixes? (Section 6.2)
- **RQ2**: How effective is the ranking of fix candidates? (Section 6.3)

Table 1. Accuracy of predicting exactly the human fix for different kinds of bugs.

Bug category	Static analyzer	Examples	Top-1 accuracy		Top-5 accuracy	
NullPointerException	Infer	804	94	(12%)	156	(19%)
BoxedPrimitiveConstructor	Error Prone	260	236	(91%)	238	(92%)
ClassNewInstance	Error Prone	21	6	(29%)	11	(52%)
DefaultCharSet	Error Prone	55	28	(51%)	42	(76%)
OperatorPrecedence	Error Prone	49	6	(12%)	30	(61%)
ReferenceEquality	Error Prone	79	22	(28%)	53	(67%)
Total		1,268	381		526	

- **RQ3**: How does Getafix compare to simpler variants of the approach? (Section 6.4)
- **RQ4**: How does the amount of available training data affect the approach? (Section 6.5)
- **RQ5**: How efficient is the approach? (Section 6.6)
- **RQ6**: How effective is Getafix in an industrial software development setting? (Section 6.7)

6.1 Experimental Setup

To address RQ1 to RQ5, we evaluate Getafix with pairs of *before* and *after* Java files that address six kinds of problems. Table 1 lists the bug categories and the static analysis tool that detects instances of these categories. We use bugs detected by Infer[3] [Calcagno et al. 2015] and Error Prone [Aftandilian et al. 2012], two state-of-the-art static analyzers for Java. To gather fixes of potential NullPointerExceptions, we gather fixes made by developers at Facebook as a reaction to warnings reported by Infer. For the other five bug categories, we gather fixes made by developers of open-source developers by searching for commits mentioning those warnings. Since some commits also contain code changes unrelated to fixing the specific bug category, we semi-automatically split and filter commits into pairs of *before* and *after* files, so that each pair fixes exactly one instance of the bug category. Overall, the data set contains 1,268 pairs of files.

Unless otherwise mentioned, all experiments are performed in a 10-fold cross-validation setup. We configure the pattern learning step to drop fix patterns representing less than 1% of the training set, to avoid polluting the prediction with very uncommon patterns.

6.2 Effectiveness in Finding Human-like Fixes

We measure Getafix's effectiveness in finding human-like fixes by checking whether a predicted fix exactly matches the known human fix. This comparison neglects empty lines but is otherwise precise, so even the addition of a comment or different whitespace choices are judged as a mismatch. The rationale is that we want to make sure the suggestions are acceptable to developers. This measure of success underapproximates the effectiveness of Getafix, because there may be other fix suggestions that a developer would have accepted than exactly the fix that the developer has applied, e.g., a semantically equivalent fix. We measure both top-1 accuracy, i.e., the percentage of fixes that Getafix predicts as the top-most suggestion, top-5 accuracy, i.e., the percentage of fixes that Getafix predicts as one of the first five suggestions.

Table 1 shows the accuracy results for all six bug categories. Depending on the bug category, Getafix suggests the correct fix as the top-most suggestion for 12% to 91% of all fixes, fixing 381 out of a total of 1,268 bugs. The top-5 accuracy is even higher, ranging between 19% and 92%, fixing a total of 526 out of 1,268 bugs.

[3] https://code.fb.com/developer-tools/open-sourcing-facebook-infer-identify-bugs-before-you-ship/

```
@Override
public String toString() {
  return new Integer(key).toString();
}
```

↦

```
@Override
public String toString() {
  return Integer.valueOf(key).toString();
}
```

(a) Fix for warning about boxing a primitive value into a fresh object.

```
// Ensure the text hasn't changed.
assert view.getText().toString()
    == metadata.getString()
    : "The text '"
    + view.getText().toString()
    + "' has been modified"
```

↦

```
// Ensure the text hasn't changed.
assert view.getText().toString().equals(metadata.getString())
    : "The text '"
    + view.getText().toString()
    + "' has been modified"
```

(b) Fix for warning about comparison with reference equality.

Fig. 9. Correctly predicted fixes for BoxedPrimitiveConstructor and ReferenceEquality warnings.

The differences between bug categories reflect how diverse and complex typical fixes for the different kinds of static analysis warnings are. For the bug category with highest accuracy, BoxedPrimitiveConstructor, there exists a relatively small set of recurrent fix patterns, which Getafix easily learns and applies. In contrast, the problem of automatically fixing potential NullPointerExceptions is harder, as developers use a wide range of fix patterns. For example, there are at least two fundamentally different ways to address a null deference warning: Either the developer recognizes that the blamed expression may indeed be null at runtime, or the developer is certain that, for reasons inaccessible to the static analyzer, the expression cannot be null. Depending on this decision, the fix will either change logic or control flow to behave reasonably in case of a null value, as, e.g., shown in Figure 1 or add a non-null assertion that convinces the static analyzer about non-nullability. Both cases provide a range of different fix patterns, which Getafix learns and applies successfully in some, but not all cases. Another factor influencing the accuracy of Getafix is, as for any learning-based approach, the amount of available training data, which we discuss separately in Section 6.5.

6.2.1 Examples of Correctly Predicted Fixes. To illustrate the strengths and limitations of Getafix, let us consider some examples. Figure 1 from the introduction gives three examples of fixes for NullPointerExceptions that Getafix predicts as the top-most suggestion. These examples illustrate that the approach not only predicts a wide range of different strategies for fixing bugs, but also selects the most suitable strategy for the given code. For example, the first fix in Figure 1, which adds a check to an existing conditional, makes sense only when the code surrounding the bug already has a conditional. The hierarchy of fix patterns, many of which provide some code context, allows our ranking to prioritize the most suitable patterns.

Figure 9 shows correctly predicted fixes for two other kinds of bug categories. The fix in Figure 9a addresses a warning about boxing a primitive value into a fresh object, which is inefficient compared to the fixed code that implicitly uses a cached instance of frequently used values. The fix in Figure 9b replaces an expression that compares two objects via reference equality with a call to `equals`. Getafix learns these fix patterns because fixes for these bug categories tend to be repetitive. Yet, finding an appropriate fix is non-trivial because there are different fix variants. For example, other fixes for the BoxedPrimitiveConstructor warning address different combinations of wrapped types, e.g., floats or strings, and different types of object wrappers, e.g., `Float` or `Boolean`. Likewise, other fixes

```
// ...
bar = x.foo();
// ...
```

↣

```
// ...
if (x == null) throw new IllegalStateException("custom message");
bar = x.foo();
// ...
```

(a) Fix for potential null deference.

```
// ...
return clazz.cast(in.newInstance());
// ...
// ...
```

↣

```
import java.lang.reflect.InvocationTargetException;
// ...
  return clazz.cast(in.getConstructor().newInstance());
// ...
} catch (InvocationTargetException e) {
} catch (NoSuchMethodException e) {
// ...
```

(b) Fix for ClassNewInstance warning.

Fig. 10. Fixes that Getafix could not predict correctly.

for the ReferenceEquality warning use an inequality check instead of an equality check, they might use `Objects.equals()` to do the equality check, or they may be part of a more complex expression.

6.2.2 Examples of Missed Fixes. To better understand the limits of Getafix, Figure 10 shows two fixes that the approach does not find. Figure 10a fixes a potential null dereference by throwing an IllegalStateException in case a variable is null. While finding this additional statement is well in scope Getafix, it fails to predict the custom error message that is passed as a string to the IllegalStateException. Since the message is application-specific, learning to predict it is a hard problem. Figure 10b fixes a warning about the `newInstance` API, which is discouraged because it bypasses exception checking. Fixes of this problem often have to adapt the exception handling around the actual fix location. Since these adaptations heavily depend on the existing exception handling, Getafix often fails to predict exactly the correct fix. This limitation could likely be mitigated by a larger set of training examples, which would enable Getafix to learn popular exception handling scenarios. Other reasons why Getafix misses fixes include fixes that rename variables, simply remove buggy code, or that address a deeper root cause of the problem.

6.3 Effectiveness of Ranking

A crucial component of Getafix is the ranking of candidate fixes, which enables the approach to decide on a small number of fix suggestions, without passing all candidates to any expensive validation step. Figure 11 evaluates the effectiveness of the ranking, showing as a blue line ("our learning, our prediction") how many of all fixes Getafix predicts correctly within the k highest ranked suggestions per bug. The percentage shown on the right of each plot, with $k = \infty$, indicates for how many bugs the correct fix is somewhere in the entire set of fix candidates. The results show that Getafix's ranking effectively pushes the correct fix toward the top of all candidate fixes, covering a large fraction of all fixes it can find within the first few suggestions.

6.4 Ablation Study

To evaluate the effectiveness of both our hierarchical learning algorithm and our ranking technique, we compare them against baseline approaches. As a baseline for learning, we use the greedy clustering algorithm used by Revisar [Rolim et al. 2018]. Revisar first partitions concrete edits by

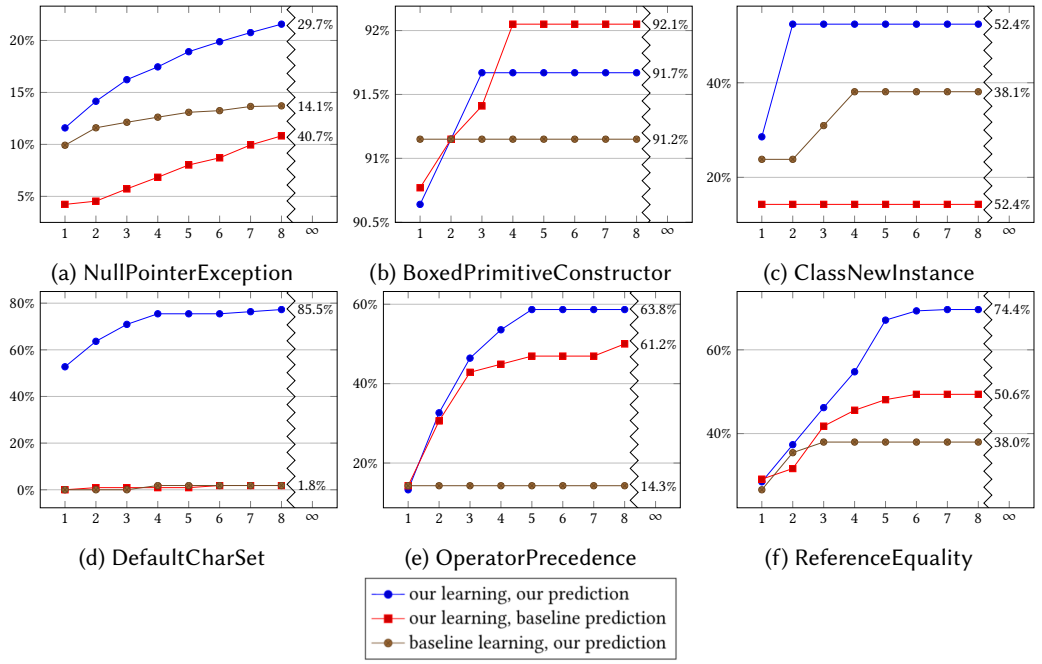


Fig. 11. Fraction of human fixes (y axis) covered by top-k (x axis) highest ranked fix candidates.

their "d-cap", which means that all concrete edits in one partition will have ASTs that are identical for all nodes up to depth $d-1$. Within each partition, Revisar then considers one edit after another, attempting to cluster them into a pattern that becomes more and more general. When the addition of an edit to a cluster would result in a pattern that contains an unbound hole, this edit instead forms a new cluster. We conservatively approximate this behavior by selecting patterns from our hierarchy of patterns that could have been found by greedy clustering if concrete edits were iterated in an ideal order. We focus our comparison on 1-cap, as it results in more general patterns than $d > 1$. We also drop additional context (see Section 4.4). As a baseline for prediction, we learn a hierarchy of fix patterns as described in Section 4.2, but then always pick the most common pattern of the hierarchy, instead of applying our ranking from Section 5.2. In case this pattern matches at multiple code locations, we apply it to the location closest to the reported issue. When evaluating the top-k accuracy of this baseline, we repeatedly pick the next most common pattern.

Figure 11 compares Getafix to the two baselines. For five out of six bug categories, both hierarchical learning and our multi-score ranking are clearly worthwhile. For BoxedPrimitiveConstructor, all three approaches perform roughly the same, which we attribute to the fact that this category has relatively few ways of fixing the bug. The most drastic drop in accuracy can be observed without the hierarchical clustering, and hence without the valuable inner nodes of the dendrogram. The DefaultCharSet category sees particularly large drop in accuracy because most human fixes involve adding an import statement but the baseline learning and prediction do not support learning and applying multiple learned patterns together.

6.5 Influence of Available Training Data

To measure the influence of the amount of available training data, we run Getafix with subsets of the 804 human fixes for null dereference bugs. Figure 12 shows the accuracy of reproducing

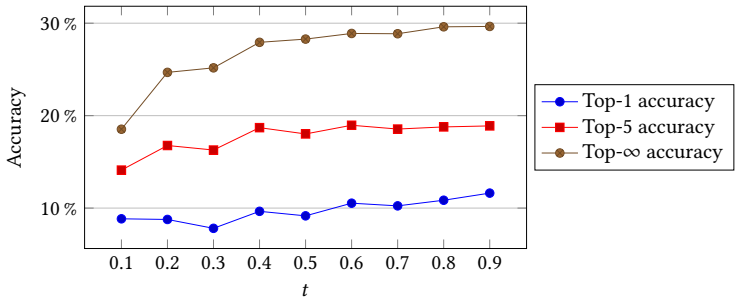


Fig. 12. Accuracy of top-1, top-5 and top-∞ predictions (fraction of predicted human NullPointerException fixes) depending on fraction t of data used for training.

Table 2. Time taken by Getafix for 10-fold experiment of training and prediction.

Bug category	Examples	Training time	Prediction time	Single prediction
NullPointerException	804	762s	3132s	3.9s
BoxedPrimitiveConstructor	260	164s	442s	1.7s
ClassNewInstance	21	60s	195s	9.2s
DefaultCharSet	55	76s	240s	4.4s
OperatorPrecedence	49	70s	138s	2.8s
ReferenceEquality	79	114s	563s	7.2s

the human fix among top-k fix suggestions, depending on the fraction t of data used for training. We use for validation the samples not used for training, and repeat the experiment for each t such that each fix was used at least eight times for both training and validation. As expected for a learning-based approach, more training data yields higher accuracy. The reason is that Getafix learns more patterns, as well as more accurate and representative metadata for those patterns (frequency, line distance information, etc.), which improves the ranking.

6.6 Efficiency

Table 2 lists the total time spent on training and prediction for each 10-fold experiment. A more realistic setup would correspond to one fold of this experiment. Breaking down the total time to individual predictions of fixes (last column), we find that Getafix takes between 1.7 seconds and 9.2 seconds on average to predict a fix. Most of this time is spent on formatting each candidate fix through a third-party library. We do so since the AST transformation can theoretically result in sub-optimal formatting, which can slightly influence ranking. Since taking a few seconds is acceptable for us, we have not attempted to optimize this further. The experiments where conducted on a single machine with 114GB of RAM and Intel Skylake processors with 24 cores running at 2.4GHz.

6.7 Real-World Deployment: Auto-Fix Suggestions for Infer Bug Reports

Getafix is deployed in Facebook to automatically suggest fixes for null dereference bugs reported by Infer. For this deployment, we dropped all patterns that add assertions to the code or that just remove code, because these patterns produce a plethora of candidates that are hard to rank. Before displaying auto-fix suggestions, we rerun Infer to ensure that previously reported warnings have disappeared. Getafix performs this validation only for the top-ranked predicted fix, which is the only one shown to the developer (if the validation passes); showing additional fix candidates would

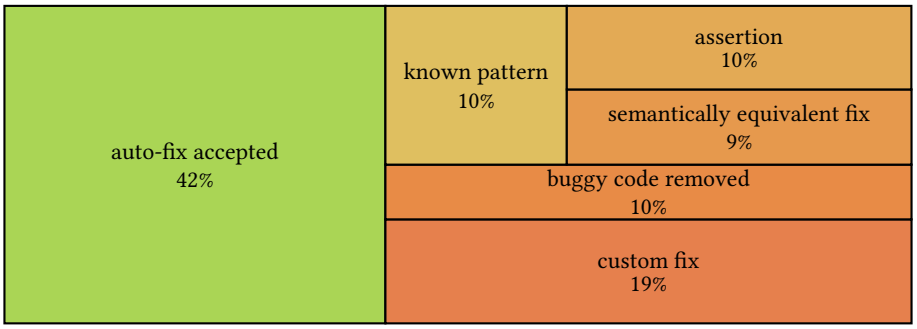


Fig. 13. Reactions to about 250 null deference warnings for which Getafix suggested a fix.

require additional runs of Infer, and besides, may be distracting for the developer. For 84% of these validation runs, the top-most predicted fix removes the previously reported Infer warning, and Getafix hence suggests the fix to the developers.

Within three months of deployment, developers addressed around 250 null dereference warnings that we showed a fix suggestion for. Figure 13 categorizes the reactions of developers: The developers directly accepted 106 suggested fixes (42%). These fixes include many non-trivial code changes (e.g., see Figure 1), i.e., Getafix helped save precious developer time. Remarkably, the acceptance rate of 42% is significantly higher than the top-1 accuracy during the experiments reported in Section 6.2. This difference is because developers are likely to accept more than one possible fix for a given bug, whereas Section 6.2 compares the predicted fix against a single known human fix.

For those fix suggestions that were not immediately accepted, Figure 13 categorizes them by their potential to be claimed by Getafix. In 10% of the cases, the developers eventually wrote a fix that Getafix knows, but that it did not rank highest and hence did not suggest. Better ranking could address these cases. Another 10% of the time, the developer added an assertion that prevents the static analysis warning. As indicated earlier, we have intentionally not rolled out such patterns so far, to not encourage developers to suppress analysis warnings. Around 9% of the time, a semantically equivalent fix to that suggested by Getafix was written. For example, for a bug in an `else` block, the developer turned the block it into an `else if`, rather than creating a new `if`. Learning more specialized patterns from more training data, or manual interpolation of learned patterns, will help to address such cases. In another 10% of the cases, the issue was resolved by just removing the code that contains the warning. Finally, the remaining human fixes were mostly custom and probably cannot be expressed as a recurring fix pattern.

In addition to warnings about null dereferences, we have deployed Getafix for two other Infer warnings, Field Not Nullable and Return Not Nullable. The acceptance rate of fixes suggested by Getafix is around 60% for both warnings. Moreover, displaying an auto-fix next to a warning resulted in a 4% and 12% higher fix rate, respectively. This amounts to around 80 additionally fixed warnings (across both warning types) per month. We also observed that Return Not Nullable warnings are usually fixed around twice as fast if an auto-fix is displayed.

Overall, we conclude from our experience of deploying Getafix that automatic repair can be beneficial in practice. To the best of our knowledge, this is the first industrially deployment of an automated repair tool that automatically learns fix patterns. Two take-aways for what has been crucial for making Getafix useful in Facebook are (i) to integrate auto-fixes into existing development tools, e.g., by showing them in the code-review tool, and (ii) to predict fixes fast enough to not slow down the development process, e.g., by ensuring that suggestions do not add much to the time a developer waits anyway for the static analysis reports.

7 LIMITATIONS

Getafix is great at finding fixes where all the necessary context is present near the bug in question. However, any class of bug for which context necessary to create a fix is located far away from the snippet of code with the bug (e.g. fix depending on definition in superclass) will be difficult to learn repairs for and therefore we do not handle such classes of bugs at this time. We found that a large fraction of bugs do in fact have all necessary context in immediate proximity, and as such we have chosen to focus on these bugs for now. In section 6.2.2 and Figure 10 we show and discuss examples of fixes Getafix was not able to produce.

Additionally, the patches that Getafix produces may produce unexpected runtime behavior. While bug fixes often have to be semantics-changing (since the current behavior is buggy), sometimes the new semantics may be undesirable. Avoiding this would require a complete specification of the expected behavior, which we do not have in practice. In the services we rolled out this rarely happens, but an example would be fixing `NullPointerExceptions` by inserting an "early return" into a function: In case a new bug instance has high similarity with training data that inserts an "early return", the pattern may be chosen. However, there is no guarantee that this adjustment of control flow is acceptable in this case. To be safe, we rely on a human reviewer to judge whether a fix introduces unexpected behavior. We could reduce this risk by also taking test suites into account, but have not done so thus far as that adds to the cost of validation.

8 RELATED WORK

8.1 Automated Program Repair

Getafix has goals similar to those of existing automated program repair techniques [Goues et al. 2019], but fills a previously unoccupied spot in the design space. In contrast to generate-and-validate approaches, such as GenProg [Goues et al. 2012], we focus on learning patterns from past fixes for specific kinds of warnings. Specifically, Getafix does not attempt to find generic solutions from any sort of ingredient space, or by generically mutating the code. One additional way in which Getafix differs from generate-and-validate repair systems is that Getafix relies on perfect fault localization implicit in the error report from the static analyzer; by contrast, most generate-and-validate repair systems include a phase of (generally) statistical fault localization as their first step. Uncertainty in fault localization would likely compromise the accuracy of Getafix, as has been acknowledged for generate-and-validate systems as well.

While Genesis [Long et al. 2017] follows a similar approach of learning transformations between *before* and *after* ASTs, its learned templates contain generators, which increase the size of the search space. Genesis-style generators would allow multiple of our edit patterns to be represented by a single pattern, but at the expense of more time spent exploring the space of fixes at generation time. Instead, patterns learned by Getafix are more specific, enabling it to often find a human-like fix with a one-shot approach. Therefore, if Genesis was trained to resolve a narrow class of bugs just like we train Getafix, Genesis would likely produce fixes similar to ours, but due to the larger search space it would not produce them as quickly. SketchFix [Hua et al. 2018] also aims at validating as few fix candidates as possible, but relies on runtime information from repeated test executions. In contrast, Getafix does not use any tests. Kim et al. [2013] automatically group human-written fixes by similarity. However, inspecting these groups and creating fix patterns from them remains a manual step, whereas Getafix learns fix patterns fully automatically. History-driven program repair [Le et al. 2016] has a diffing and mining pipeline similar Getafix, but uses it to infer abstract edit steps (like "insert statement"), which then help rank fix candidates produced by a generate-and-validate repair technique. Prophet learns a generic model of how natural a fix is and uses it for ranking fix candidates [Long and Rinard 2016]. Their approach could be plugged into

Getafix, as an alternative or addition to our ranking algorithm. Section 5.3 discusses their ranking in more detail. CapGen [Wen et al. 2018] uses AST context to select mutation operators and fixing ingredients. Getafix implicitly uses a similar prioritization using its clustering and ranking strategy.

Soto et al. [2016] observe that fix patterns differ across programming languages and analyze the structure of typical patterns in Java. Martinez et al. [Martinez and Monperrus 2012, 2015, 2018] also performed extensive AST analysis on Java repositories to statistically analyze code change patterns, which can guide program repair. Soto and Goues [2018] propose a probabilistic model-based approach for Java, which produces higher quality fixes more likely. As Getafix works on the level of ASTs and learns from human fixes, it can learn language-specific fix patterns automatically. NPEfix [Cornu et al. 2015] addresses NullPointerExceptions by selecting at runtime from a set of manually defined strategies, such as skipping the statement or replacing the null values with a fresh object. In contrast, Getafix addresses arbitrary bug categories and applies fixes statically.

8.2 Mining of Edit Patterns

Refazer [Rolim et al. 2017] uses PROSE [Polozov and Gulwani 2015] and a DSL of program transformations to learn repetitive edits. Revisar [Rolim et al. 2018] influenced Getafix by also using anti-unification [Kutsia et al. 2014] to derive edit patterns from concrete edits, but based on greedy clustering and without extracting surrounding context. Another difference is that Revisar learns from arbitrary code changes, not fixes of specific bug categories, requiring manual selection of interesting patterns from hundreds of candidates. See Sections 4.4 and 6.4 for a detailed comparison. Brown et al. [2017] mine recurring bug-introducing changes as the reverse operations of commits. Their goal is to infer mutation operations, i.e., code changes that inject artificial bugs into arbitrary code, whereas Getafix aims at fixing bugs.

8.3 Machine Learning on Code

Several approaches delegate parts of or the entire fix generation task to a neural network. SSC combines a set of manually written generators of fix candidates with learned models that decide which candidate to apply [Devlin et al. 2017]. Sarfgen uses embeddings of code to formulate repair as a search for a similar reference solution [Wang et al. 2018]. Instead, Getafix aims at suggesting fixes for newly developed software that has no reference solution. DeepFix learns an end-to-end model for predicting fixes of compilation errors [Gupta et al. 2017]. Combining end-to-end learning with our idea of focusing on fixes for particular bug categories may be interesting future work.

Getafix relates to a stream of work on machine learning applied to code [Allamanis et al. 2018]. Pradel and Sen [2018] demonstrate the power of identifier names for reasoning about buggy code. A similar technique could benefit our learning and ranking phase, e.g., by using word embeddings of identifiers and literals to guide which variables to substitute holes with, or by mimicking human intuition about return values to pick in case of an early return. Work on learning to represent code edits may also provide a starting point for learning to suggest fixes [Yin et al. 2018].

9 CONCLUSION

Fixing incorrect code is a long-standing problem that consumes significant developer time. Fortunately, many fixes, in particular bugs identified by static analyzers, are instances of recurring patterns. This paper presents Getafix, the first industrially deployed automated repair tool that fully automatically learns fix patterns from past, human-written commits. The approach is based on a novel hierarchical, agglomerative clustering algorithm, which summarizes a given set of fix commits into a hierarchy of fix patterns. Based on this hierarchy and a ranking technique that decides which fix pattern is most appropriate for a new occurrence of a bug category, Getafix proposes a ranked list of fixes to the developer. An evaluation with 1,268 real-world bug fixes and

our experience of deploying Getafix within Facebook show that the approach accurately predicts human-like fixes for various bugs, reducing the time developers have to spend on fixing recurring kinds of bugs.

We believe that the potential of the Getafix approach is broader than fixing only static analysis bugs. As long as the bug category and bug location is known, and a training set of fixes is available, Getafix can offer fixes learned from the training set. Case in point, Getafix suggests fixes — via SapFix[4] [Marginean et al. 2019] — for null pointer exceptions detected by Sapienz[5], an automated testing system for mobile apps.

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 a classifier that provides an alternative approach for ranking between fix patterns.

REFERENCES

Edward Aftandilian, Raluca Sauciuc, Siddharth Priya, and Sundaresan Krishnan. 2012. Building Useful Program Analysis Tools Using an Extensible Java Compiler. In *12th IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2012, Riva del Garda, Italy, September 23-24, 2012*. 14–23.

Miltiadis Allamanis, Earl T Barr, Premkumar Devanbu, and Charles Sutton. 2018. A survey of machine learning for big code and naturalness. *ACM Computing Surveys (CSUR)* 51, 4 (2018), 81.

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

David Bingham Brown, Michael Vaughn, Ben Liblit, and Thomas W. Reps. 2017. The care and feeding of wild-caught mutants. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2017, Paderborn, Germany, September 4-8, 2017*. 511–522.

Cristiano Calcagno, Dino Distefano, Jérémy Dubreil, Dominik Gabi, Pieter Hooimeijer, Martino Luca, Peter O'Hearn, Irene Papakonstantinou, Jim Purbrick, and Dulma Rodriguez. 2015. Moving fast with software verification. In *NASA Formal Methods Symposium*. Springer, 3–11.

Maria Christakis and Christian Bird. 2016. What developers want and need from program analysis: an empirical study. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ASE 2016, Singapore, September 3-7, 2016*. 332–343. https://doi.org/10.1145/2970276.2970347

Benoit Cornu, Thomas Durieux, Lionel Seinturier, and Martin Monperrus. 2015. Npefix: Automatic runtime repair of null pointer exceptions in java. *arXiv preprint arXiv:1512.07423* (2015).

Jacob Devlin, Jonathan Uesato, Rishabh Singh, and Pushmeet Kohli. 2017. Semantic Code Repair using Neuro-Symbolic Transformation Networks. *CoRR* abs/1710.11054 (2017). arXiv:1710.11054 http://arxiv.org/abs/1710.11054

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.

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.

Claire Le Goues, Michael Pradel, and Abhik Roychoudhury. 2019. Automated Program Repair. *Commun. ACM* (2019). To appear.

[4] https://code.fb.com/developer-tools/finding-and-fixing-software-bugs-automatically-with-sapfix-and-sapienz/
[5] https://code.fb.com/developer-tools/sapienz-intelligent-automated-software-testing-at-scale/

Rahul Gupta, Soham Pal, Aditya Kanade, and Shirish Shevade. 2017. DeepFix: Fixing Common C Language Errors by Deep Learning. In *AAAI*.

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.

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.

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

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.

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.

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

Alexandru Marginean, Johannes Bader, Satish Chandra, Mark Harman, Yue Jia, Ke Mao, Alexander Mols, and Andrew Scott. 2019. SapFix: Automated End-to-End Repair at Scale *(ICSE-SEIP '19)*.

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

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

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

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

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

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

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

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

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

Ke Wang, Rishabh Singh, and Zhendong Su. 2018. Search, align, and repair: data-driven feedback generation for introductory programming exercises. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2018, Philadelphia, PA, USA, June 18-22, 2018*. 481–495.

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.

Pengcheng Yin, Graham Neubig, Marc Brockschmidt Miltiadis Allamanis and, and Alexander L. Gaunt. 2018. Learning to Represent Edits. *CoRR* 1810.13337 (2018).