

NoFAQ: Synthesizing Command Repairs from Examples

Loris D’Antoni
loris@cs.wisc.edu
University of Wisconsin
Madison, USA

Rishabh Singh
risin@microsoft.com
Microsoft Research
Redmond, USA

Michael Vaughn
vaughn@cs.wisc.edu
University of Wisconsin
Madison, USA

ABSTRACT

Command-line tools are confusing and hard to use due to their cryptic error messages and lack of documentation. Novice users often resort to online help-forums for finding corrections to their buggy commands, but have a hard time in searching precisely for posts that are relevant to their problem and then applying the suggested solutions to their buggy command. We present NoFAQ, a tool that uses a set of rules to suggest possible fixes when users write buggy commands that trigger commonly occurring errors. The rules are expressed in a language called `FIXIT` and each rule pattern-matches against the user’s buggy command and corresponding error message, and uses these inputs to produce a possible fixed command. NoFAQ automatically learns `FIXIT` rules from examples of buggy and repaired commands. We evaluate NoFAQ on two fronts. First, we use 92 benchmark problems drawn from an existing tool and show that NoFAQ is able to synthesize rules for 81 benchmark problems in real time using just 2 to 5 input-output examples for each rule. Second, we run our learning algorithm on the examples obtained through a crowd-sourcing interface and show that the learning algorithm scales to large sets of examples.

CCS CONCEPTS

•Software and its engineering → Command and control languages; Programming by example;

KEYWORDS

Domain Specific Languages, Programming by Example, Program Synthesis, Program Repair, Command Line Interface

ACM Reference format:

Loris D’Antoni, Rishabh Singh, and Michael Vaughn. 2017. NoFAQ: Synthesizing Command Repairs from Examples. In *Proceedings of 2017 11th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, Paderborn, Germany, September 4–8, 2017 (ESEC/FSE’17)*, 11 pages. DOI: 10.1145/3106237.3106241

1 INTRODUCTION

Command-Line Interfaces (CLIs) let users interact with a computing system by writing sequences of commands. CLIs are especially

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ESEC/FSE’17, Paderborn, Germany

© 2017 Copyright held by the owner/author(s). Publication rights licensed to ACM. 978-1-4503-5105-8/17/09...\$15.00
DOI: 10.1145/3106237.3106241

popular amongst advanced computer users, who use them to perform small routine tasks such as committing a file to a repository with version control, installing software packages, compiling source code, finding and searching for files etc. Even though this mode of interaction has been supplanted by more natural graphical user interfaces for many common tasks, CLIs are still routinely used for most scripting tasks in Unix and Mac OS. Even the Windows operating system now officially provides complex command-line interfaces with products such as Windows Powershell.

Since command-line interactions often require complex parameters and flag settings to specify behavior, non-expert users find CLIs challenging to use. Moreover, after entering an incorrect command, the user has to deal with cryptic errors that are hard to decipher by just looking at the verbose text-based documentation of the commands. For these reasons, users typically resort to online help-forums for finding corrections to their buggy commands. Unfortunately, this can also be problematic as users need to precisely search for posts related to the issues with their commands and then transform the suggested solutions to apply them in their context.

What About Common Errors? Recently, `THEFXXX`¹ was developed to automatically address common errors when working with a CLI. If after typing a command a user receives an error message, `THEFXXX` uses a set of hard-coded rules to suggest possible fixes to the command. Typical fixes include adding missing flags, creating a missing directory, or changing file extensions. `THEFXXX` is extremely popular and, on GitHub, it has already been starred by more than 24,000 users and has been forked more than 1,200 times. Despite its success, `THEFXXX` has a significant limitation: to add a new rule a developer first needs to understand the syntax and precise semantics of `THEFXXX` and then manually hard-code the rule into the tool. Due to this, newly added rules have at times caused non-terminating or unexpected behaviors².

Synthesizing Rules From Examples Inspired by the success and limitations of `THEFXXX`, we built NoFAQ (No more Frequently Asked Questions), which also uses a set of rules for fixing common errors, but it differs from `THEFXXX` in the following key aspects:

- (1) Rules are encoded in a declarative domain-specific language (DSL) called `FIXIT`.
- (2) New rules are automatically synthesized from crowd-sourced examples of buggy and repaired commands.

NoFAQ is used by novice programmers, who query it for suggestions for common errors (Figure 1), and by expert programmers, who contribute examples of fixes for commands for which users asked for hints, but NoFAQ did not have a suitable rule (Figure 2). Unlike `THEFXXX`, NoFAQ can add rules in a completely unsupervised fashion and does not require contributor access to source code

¹We censored the name of the tool <http://bit.ly/CmdCorrection>.

²<http://bit.ly/1j7zxOr> and <http://bit.ly/1YgngXJ>.

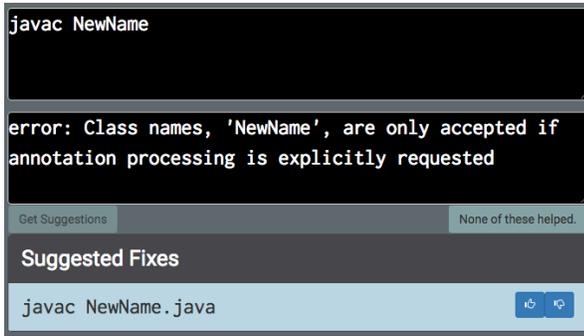


Figure 1: Interface to get suggestions.

or know a complex framework. In fact, while THEFXXX only consists of fewer than 100 rules in a little over 1 year, NoFAQ already contains hundreds of rules and constantly learns new ones.

The FIXIT DSL for encoding fix rules is inspired by the types of rules appearing in THEFXXX and by common command repairs requested by users on help-forums. A FIXIT rule first uses pattern matching and unification to match the command and error message, and then applies a fix transformation if the match succeeds. The transformations consist of substring and append functions on strings present in the command and error message.

We present an algorithm that efficiently synthesizes FIXIT rules consistent with a given set of input-output examples using a Version-space Algebra (VSA) [12]. VSA-based synthesis techniques are used to succinctly represent the set of all expressions that are consistent with a set of examples [7]. Even though existing VSA data structures can represent an exponential number of FIXIT rules in polynomial space, this space can still be quite large. To address this problem we introduce *lazy version-space algebra*, which is inspired from Plotkin's [22] work on using least general generalization for unifying two logical predicates. Given a set of examples, our algorithm maintains a lazy representation of a subset of all FIXIT rules consistent with the examples. The missing rules are only enumerated when necessary—i.e., when a new input-output example can only be accounted for by adding a FIXIT rule that is not already present in the version-space. Because of the careful design of FIXIT, our synthesis algorithm has a polynomial time complexity. In contrast, existing VSA-based synthesis techniques for string transformations require exponential time [7]. The polynomial time complexity is crucial, allowing our synthesis algorithm to scale to a large number of fix examples.

We evaluate the synthesis algorithm implemented in NoFAQ on 92 benchmark problems obtained from both THEFXXX (76) and online help-forums (16). NoFAQ is able to learn the repair rules for 81 of the buggy commands in these benchmark problems from only 2 to 5 input-output examples each. We also evaluate the algorithm on crowd-sourced examples and show that, on average, we process a new example in less than .37 seconds.

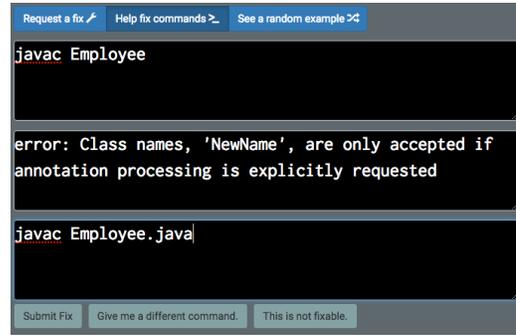


Figure 2: Interface to crowd-source examples of fixes.

Contributions summary:

- (1) NoFAQ, a tool for learning rules for fixing common errors using crowd-sourced examples (§ 3)
- (2) FIXIT, a domain-specific language for encoding rules that map a command and an error message to possible fixed commands
- (3) A sound and complete polynomial time synthesis algorithm for FIXIT rules based on lazy version-space algebra (§ 4 and § 5).
- (4) A qualitative and quantitative evaluation of the synthesis algorithm on 92 benchmarks obtained from both THEFXXX and online help-forums (§ 6).

2 MOTIVATING EXAMPLES

We first present the main ideas behind NoFAQ using concrete examples appearing in THEFXXX system.

2.1 Adding Missing File Extension

Novice Java programmers are likely to encounter the following error when accidentally passing a class name instead of a source code file to the `javac` compiler:

```
cmd1: javac Employee
err1: Class names, 'Employee', are only accepted if annotation
      processing is explicitly requested
```

A novice user, who does not know how to proceed, can use the NoFAQ interface to ask for a fix. If no fix is available, NoFAQ will add this input to the ones for which crowd-sourcing is needed. A seasoned programmer who uses the crowd-sourcing interface of NoFAQ (Figure 2) would immediately recognize the problem and add the extension `.java` at the end of the input file.

```
fix1: javac Employee.java
```

Let's assume that another user submits a similar query and a skilled developer provides the following fix for it.

```
cmd2: javac Pair
err2: Class names, 'Pair', are only accepted if annotation pro-
      cessing is explicitly requested
fix2: javac Pair.java
```

NoFAQ synthesizes the following fix rule from the two examples:

```
match [STR(javac), VAR-MATCH(1,  $\epsilon$ ,  $\epsilon$ )]
and [STR(Class), STR(names, ), VAR-MATCH(2, ' ', ), STR(are),
STR(only), STR(accepted), STR(if), STR(annotation),
STR(processing), STR(is), STR(explicitly), STR(requested)]
→ [FSTR(javac), SUB(0, 0,  $\epsilon$ , . java, VAR(1))]
```

The first part of the rule (i.e., up to the symbol \rightarrow) pattern-matches against the command and the error message and binds the input strings to corresponding variables, which are then used by the second part of the rule to produce the output. In this case the $\text{SUB}(0, 0, \epsilon, . \text{java}, \text{VAR}(1))$ expression extracts the complete string associated with $\text{VAR}(1)$ (a start index of 0 and an end index of 0 denotes the identity string extraction), and then prepends the string ϵ at the beginning, and appends the string $. \text{java}$ at the end. If another user submits a query for a similar mistake, NoFAQ will be able to provide a suggestion for it using this rule (Figure 1).

2.2 Extracting Complex Substrings

In the following example, a user trying to move a picture from one location to another gets the following error message, which is addressed in NoFAQ with the proper fix.

cmd1: mv photo.jpg Mary/summer12.jpg
err1: can't rename 'photo.jpg': No such file or directory
fix1: mkdir Mary && mv photo.jpg Mary/summer12.jpg

Given this example and another similar one, NoFAQ synthesizes the following rule.

```
match [STR(mv), VAR-MATCH(1,  $\epsilon$ ,  $\epsilon$ ), VAR-MATCH(2,  $\epsilon$ ,  $\epsilon$ )]
and [STR(can't), STR(rename), VAR-MATCH(3, ' ', ),
STR(No), STR(such), STR(file), STR(or), STR(directory)]
→ [FSTR(mkdir), SUB(0, Cp(/, 1, 0),  $\epsilon$ , &&, VAR(2))
FSTR(mv), SUB(0, 0,  $\epsilon$ ,  $\epsilon$ , VAR(1)), SUB(0, 0,  $\epsilon$ ,  $\epsilon$ , VAR(2))]
```

The second expression in the output extracts the directory name—i.e., the substring that starts at index 0 and ends at the index of first occurrence of the character /. The rule also adds a string && at the end of the extracted string to sequence the two output commands.

3 THE NOFAQ SYSTEM

In this section, we briefly describe the NoFAQ system and the ways users interact with it. NoFAQ is structured as a web app with three main sections: 1) a section that allows users to request suggestions on how to fix erroneous commands, 2) a section that allows skilled users to contribute example fixes to previously requested commands, and 3) a section that allows users to view existing examples of fixes. We describe the features of the three sections and how they interact with our synthesis algorithm.

To request a fix, a user simply enters an erroneous command with the corresponding error. If NoFAQ has already learned some rules matching the input, the corresponding outputs are presented to the user (Figure 1).³ The user can decide to upvote some fix or flag some of the suggested fixes as bad ones. If no rule matches, or the user clicks that none of the suggestions was helpful, the

command and error are added to the set of inputs for which we need to crowd-source an example fix. Skilled users can then use the second section to contribute fixes to such inputs.

In the second section, a user is prompted with a random input for which NoFAQ currently does not have a fix. The user can 1) provide a fix for it, 2) ask NoFAQ to provide a different random input, or 3) flag the input as not fixable (e.g., there is not enough information to provide a meaningful fix). If two users provide the same fix for a given input, the example is added to the database of NoFAQ and is used by our synthesis algorithm to learn new rules.

In the third section, users can see random examples of fixes provided by other users and upvote some fix or flag some fixes as bad ones. This mechanism mitigates the problem of malicious users trying to provide bad fixes. To further address the problem, we also use a blacklisting mechanism to remove all inputs and fixes that contain profanity or typical bad fixes like `rm -rf /*`.

4 SYNTHESISING RULES IN NOFAQ

In this section, we first describe FIXIT, a domain-specific language for expressing repair rules and then present an algorithm for synthesising FIXIT rules from examples.

4.1 The FIXIT Language

The syntax of FIXIT is presented in Figure 3. The FIXIT language is designed to be expressive enough to capture most of the rules we found in THEFXXX and in online help-forums, but also concise enough to enable efficient learning from examples.

General structure Each FIXIT program is a rule of the form

$$\text{match } cmd \text{ and } err \rightarrow fix$$

that takes as input a command \bar{s}_{cmd} and an error \bar{s}_{err} and either produces a fixed command or the undefined value \perp . The inputs \bar{s}_{cmd} and \bar{s}_{err} are lists of strings that are obtained by extracting all space-separated strings appearing in the input command and error message respectively. The output fix produced by the rule is also a list of strings. From now on, we assume that the inputs and outputs are lists of strings that do not contain space characters.

A rule has 3 components: 1) A list of match expressions $cmd = [m_1, \dots, m_l]$ used to pattern match against the input command \bar{s}_{cmd} . 2) A list of match expressions $err = [m_1, \dots, m_k]$ used to pattern match against the input error message \bar{s}_{err} . 3) A list of fix expressions $fix = [f_1, \dots, f_n]$ to produce the new fixed command.

Match expressions A match expression m is either of the form $\text{STR}(s)$ denoting a constant string s or of the form $\text{VAR-MATCH}(i, l, r)$. A $\text{VAR-MATCH}(i, l, r)$ expression denotes a variable index i and requires the matched string to start with the prefix l and end with the suffix r . We assume that no two variable expressions appearing in the match expression have the same index. When a list of match expressions $[m_1, \dots, m_l]$ is applied to a list of strings $\bar{s} = [s_1, \dots, s_l]$ with the same length l , it generates a partial function $\sigma : \mathbb{N} \mapsto \Sigma^*$ that assigns variables appearing in the match expressions to the corresponding strings in the input. For example, evaluating the expression

$$[\text{STR}(mv), \text{VAR-MATCH}(1, \epsilon, . \text{jpg}), \text{VAR-MATCH}(2, \epsilon, . \text{jpg})]$$

³After observing that many users were not providing the error message or only parts of it, we allow the provided error to match a substring of the error in the rule.

Fix rule	r	$:=$	match <i>cmd</i> and <i>err</i> \rightarrow <i>fix</i>
Input cmd	<i>cmd</i>	$:=$	$[m_1, \dots, m_l]$
Input error	<i>err</i>	$:=$	$[m_1, \dots, m_k]$
Output cmd	<i>fix</i>	$:=$	$[f_1, \dots, f_n]$
Match expr	<i>m</i>	$:=$	STR(<i>s</i>) VAR-MATCH(<i>i</i> , <i>s_l</i> , <i>s_r</i>)
Fix expr	<i>f</i>	$:=$	FSTR(<i>s</i>) SUB(<i>p_L</i> , <i>p_R</i> , <i>s_l</i> , <i>s_r</i> , VAR(<i>i</i>))
Pos expr	<i>p</i>	$:=$	IP(<i>k</i>) CP(<i>c</i> , <i>k</i> , δ)
	<i>s</i> , <i>s_l</i> , <i>s_r</i>	$:$	<i>string</i>
	<i>i</i> , <i>k</i> , δ	$:$	<i>integer</i>
	<i>c</i>	$:$	<i>character</i>

Figure 3: Syntax of the rule description language FIXIT.

on the list of strings $[mv, a.jpg, b.jpg]$ produces the function σ such that $\sigma(1) = a.jpg$ and $\sigma(2) = b.jpg$. On the other hand, evaluating the same expression on $[mv, a.png, b.jpg]$ yields \perp , as *a.png* does not match the required suffix in VAR-MATCH(1, ϵ , $.jpg$).

Fix expressions A fix expression *f* is either of the form FSTR(*s*) denoting the constant output string *s*, or of the form SUB(*p_L*, *p_R*, *s_l*, *s_r*, VAR(*i*)) denoting a function that is applied to the string *s_i* matched by the variable VAR(*i*). This function outputs the string *s_l* · *m* · *s_r*, where · denotes the string concatenation operator and *m* = substr(*s*, *j_L*, *j_R*) where *j_L* and *j_R* are the indices resulting from respectively evaluating the position expressions *p_L* and *p_R* on the string *s*. Here, given a string $s = a_0 \dots a_n$, the expression substr(*s*, *j_L*, *j_R*) denotes the string $a_{j_L} \dots a_{j_R-1}$ if $j_L, j_R \leq n + 1$ and the undefined value \perp otherwise. Notice that, unlike previous VSA-based languages [7], FIXIT does not allow binary recursive concatenation. Prohibiting this enables polynomial time synthesis.

Positions expressions A position expression *p* can either be a constant or a symbolic position. A constant position expression IP(*k*), which denotes the index *k* if *k* is positive and the index $|s| + k$ if *k* is negative. If *k* = 0, this expression evaluates to 0 when evaluated for *p_L* (i.e., the starting index of the substring) and to $|s|$ when evaluated for *p_R*, where $|s|$ denotes the length of the string *s*. For example, in the function SUB(IP(0), IP(0), ϵ , ϵ , VAR(1)), where $\sigma(1) = \text{File}$, the first constant position evaluates to the index 0, while the second constant position evaluates to the index $|\text{File}| = 4$. A symbolic position expression CP(*c*, *k*, δ), which denotes the result of applying an offset δ to the index of the *k*-th occurrence of the character *c* in *s* if *k* is positive, and the result of applying an offset δ to the index of the *k*-th to last occurrence of the character *c* in *s* if *k* is negative. For example, given the string *www.google.com*, the expression CP($.$, 1, -2) denotes the index 2 (two positions before the first dot), while the expression CP($.$, -1 , 2) denotes the index 12 (two positions after the last dot). This operator is novel and can express operations that are not supported by previous VSA-based work. In particular, FlashFill [7] only allows the extraction of the exact position of a character and not positions in its proximity. Despite this additional capability, FIXIT programs can be synthesized in polynomial time.

Comparison with FlashFill DSL At the top-level, FIXIT consists of match expressions over commands and error messages, which perform pattern-matching and unification of variables with strings. This form of matching and unification is not expressible in FlashFill, so we cannot use it to learn the fix rules directly. However, we can use FlashFill as a subroutine to learn string transformations

corresponding to expressions similar to SUB expressions in No-FAQ. However, the FlashFill DSL has two major limitations: 1) No support for offsets from regular expression matches in computing position expressions (in contrast to FIXIT's CP(*c*, *k*, δ) operator), and 2) A finite hard-coded token set for regular expressions (e.g. no support for constant character tokens). Moreover, as described in Subsection 4.2 and Section 7, our SUB operator yields a synthesis algorithm that operates in polynomial time in the number of examples, which enables the algorithm to scale to a large number of examples. Because of the recursive binary concatenation expressions in FlashFill, the DAG intersection based synthesis algorithm is exponential in the number of examples.

4.2 Synthesizing Rules from Examples

In this section we describe our algorithm for synthesizing a single FIXIT rule from a set of examples of concrete command fixes.

The algorithm for learning a single FIXIT rule is described in Figure 5; it takes as input a list of examples $E = [e_1, \dots, e_n]$ where each example e_i is a triple of the form $(\bar{s}_{cmd}, \bar{s}_{err}, \bar{s}_{fix})$ and outputs a symbolically represented set of FIXIT rules *R* consistent with *E*—i.e., for every rule $r \in R$ and example $e_i = (\bar{s}_{cmd}, \bar{s}_{err}, \bar{s}_{fix})$, the rule *r* outputs \bar{s}_{fix} on the input $(\bar{s}_{cmd}, \bar{s}_{err})$. The algorithm processes one input example at a time, and after processing the first *i* examples $E_i = [e_1, \dots, e_i]$ the algorithm has computed a set of rules R_i consistent with E_i . We use \perp to denote the undefined result. If at any point our algorithm returns \perp , there is no FIXIT rule that is consistent with the given set of examples.

4.2.1 Symbolic Representation of Multiple Rules. As there can be exponentially many rules consistent with the input examples, we adopt a symbolic representation of the set *R* that is guaranteed to have polynomial size. Our synthesis algorithm takes as input a list of examples *E* and outputs a symbolic rule of the form **match** *cmd* **and** *err* \rightarrow_s *fixes*, where *cmd* and *err* are lists of expressions that can consist of either constants or variables, and *fixes* = $[f_1, \dots, f_m]$ is a list of expressions that symbolically represents a set of outputs consistent with the examples *E*. Formally, each f_i in *fixes* is either a constant expression FSTR(*s*) for some *s*, or a set of substring expressions $\{su_1, \dots, su_k\}$, where each su_i is of the form SUB(*p_L*, *p_R*, *s_l*, *s_r*, VAR(*j*)). Intuitively, if we replace each set with one of the fix expressions it contains, we obtain a FIXIT rule. If each f_i contains *k* elements, this symbolic representation models k^n programs using an expression of size kn .

4.2.2 Lazy Rule Representation. The core element of our algorithm is a lazy representation of the rules that represents match and fix expressions as constants for as long as possible—i.e., until a new example shows that some parts of the rule cannot be constants. Lazy representation reduces the number of variable expressions, in turn reducing the number of substring expressions to be considered. We illustrate the idea with a concrete example. Say we are given the two examples shown in Figure 4a and 4b. After processing the first example, our algorithm synthesizes the FIXIT rule in Figure 4c in which every match expression and every fix expression is a constant. However, since we have only seen one example, we do not yet know whether some expression appearing in the match should actually be a variable match expression or whether some

cmd1: java Run.java
err1: Could not find or load main class Run.java
fix1: java Run

(a) First example.

```
match [STR(java), STR(Run.java)]
and [STR(Could), STR(not), STR(find), STR(or), STR(load),
     STR(main), STR(class), STR(Run.java))]
→s [FSTR(java), FSTR(Run)]
```

(c) Symbolic rule representation synthesized after first example.

cmd2: java Meta.java
err2: Could not find or load main class Meta.java
fix2: java Meta

(b) Second example.

```
match [STR(java), VAR-MATCH(1, ε, . java)]
and [STR(Could), STR(not), STR(find), STR(or), STR(load),
     STR(main), STR(class), VAR-MATCH(2, ε, . java))]
→s [FSTR(java), {
  SUB(IP(0), IP(-5), ε, ε, VAR(1)))
  SUB(IP(0), CP(., 1, 0), ε, ε, VAR(1)))
  SUB(IP(0), CP(., -1, 0), ε, ε, VAR(1)))
}]
```

(d) Symbolic rule representation synthesized after both examples.

Figure 4: Two input examples e_1 and e_2 and symbolic rules synthesized after processing e_1 and e_2 .

element in the fix expression should actually be a function of some variable. The main idea is that any of these possibilities can still be “recovered” when a new example shows that indeed a variable is needed. Using this, we maintain each expression as a constant until a new example shows that some expression cannot actually be a constant.

This is exactly what happens when processing the input example in Figure 4b. At this point in order to find a rule that is consistent with both examples we need to introduce a variable match as the second expression of the command match, and some function application as the second element of the fix. To do so, our algorithm applies the following operations to the previously computed rule.

- (1) All match expressions that cannot be constants are “promoted” to variable match expressions (making sure that all variable names are unique), which match on the longest shared prefix and suffix of all previously seen values at that position. The following table illustrates the idea for the case in which we try to unify the command **cmd2** in Figure 4b with the matching part of the already computed rule in Figure 4c.

rule:	STR(java)	STR(Run.java)
new-ex:	java	Meta.java
new-rule:	STR(java)	VAR-MATCH(1, ε, . java)

- (2) All the fix expressions that cannot be constants are “promoted” to SUB expressions that are consistent with the current examples and are allowed to use the variables which appear in the match expressions.

The second rule in Figure 4(d) reflects this update. The figure also shows how multiple SUB expressions are represented symbolically as a set. We describe all of these components in detail in the next section. This new notion of lazy VSA is crucial in our domain in which the input examples can be large. However, the technique is *general* and can also be applied to improve other VSA based synthesizers such as FlashFill [7].

4.2.3 Synthesis Algorithm. Given a list of input examples, the function SYNTRULES uses the first example and the function CONSTRULE to generate the symbolic rule composed only of constant operators. It then iteratively refines the rule on the remaining examples as shown in Figure 5. This second operation is done by the function REFINERULE which takes as input a symbolic rule r , one new example $(\bar{s}_{cmd}, \bar{s}_{err}, \bar{s}_{fix})$, and the list of examples E on which

//Rules consistent with input examples

```
fun SYNTRULES([e0, . . . , en])
  r ← CONSTRULE(e0)
  for 1 ≤ i ≤ n do           ▷ refine on each example ei
    r ← REFINERULE(r, [e0, . . . , ei-1], ei)
  return r
```

//Refines a rule to make it consistent with new example

```
fun REFINERULE(r, E, (scmd, serr, sfix))
  r ≡ (match cmd and err →s fixes)
  (cmd', Vc) ← FINDVARIABLES(scmd, cmd, 0)
  (err', Ve) ← FINDVARIABLES(serr, err, |scmd|)
  V ← Vc ∪ Ve
  E' ← (scmd, serr, sfix) :: E
  fixes' ← SYNTHFIX(sfix, fixes, E', V)
  return (match cmd' and err' →s fixes')
```

//Finds variables necessary to match example

```
fun FINDVARIABLES([s1, . . . , sn], [t1, . . . , tm], o)
```

//Outputs the fixes consistent with examples E and such that SUB expressions can use any variable in V . The fix component of the latest example and the fixes computed on the previous examples are also passed as input

```
fun SYNTHFIX([s1, . . . , sn], [t1, . . . , tn], e :: E, V)
```

Figure 5: Algorithm for synthesizing FIXIT rules.

every concrete rule represented by r behaves correctly. REFINERULE executes two main steps using the following functions.

The function FINDVARIABLES tries to unify the inputs \bar{s}_{cmd} and \bar{s}_{err} with the corresponding match expressions cmd and err in the symbolic rule r and generates new variable match expressions if necessary—i.e., when r contains a matching expression STR(s) but the corresponding component in the example is a string different from s . In this case, a VAR-MATCH(i, l, r) expression is generated such that i is a new variable name, and l and r are the longest prefix and suffix shared by s , respectively. When FINDVARIABLES is presented with a new \bar{s}_{cmd} or \bar{s}_{err} after a constant match expression has been ‘promoted’ to a VAR-MATCH(i, l, r) expression, the prefix and suffix are updated accordingly. FINDVARIABLES determines the longest prefix r' and suffix l' of l and r , respectively, that is

consistent with the appropriate component of the new example, and generates $\text{VAR-MATCH}(i, l', r')$.

The function SYNTHFIX uses the variables computed in the previous step to synthesize all possible fix expressions that are consistent with the list of examples $\{(\bar{s}_{cmd}, \bar{s}_{err}, \bar{s}_{fix})\} :: E$. To simplify variable naming and guarantee unique names, each variable has the index of the corresponding element in the input—i.e., $\text{VAR}(i)$ denotes the i -th string in the list $\bar{s}_{cmd} @ \bar{s}_{err}$ obtained by concatenating the command and error input lists.

Lazy pattern matching The function FINDVARIABLES , given a rule r and a new example e , iterates over the input components of the new example e and outputs the set of variables necessary to match this new example with respect to the previously computed symbolic rule r . The function SYNTHFIX , given a rule r and a list of examples E , individually synthesizes all the components f_i of the symbolic output fix expression that are consistent with E . SYNTHFIX is incremental in the sense that it tries to minimally change the original fix expression of r :

- if the i -th component t_i of the fix expression of r is a constant string consistent with the new example, then the algorithm leaves it unchanged;
- in any other case the output has to be a substring operation, and the function SYNTHSUBSTRINGS is used to compute all the possible SUB expressions that are consistent with the set of examples E .

Substring expressions During its computation, the function SYNTHFIX has to compute the set of all substrings consistent with the current variables detected by the lazy VSA and the current examples. Figure 5 omits the formal definition of this function due to space limitations, but we describe its main components. Given an example $e = (\bar{s}_{cmd}, \bar{s}_{err}, \bar{s}_{fix})$, a set of variable names V , and the index i corresponding to the element of the output sequence we are trying to synthesize, the algorithm computes the set of all substring expressions of the form $fun = \text{SUB}(p_L, p_R, s_l, s_r, \text{VAR}(j))$ that are consistent with e such that the result of applying fun to the j -th string in $\bar{s}_{cmd} @ \bar{s}_{err}$ is the i -th string in \bar{s}_{fix} . Let's assume that $|\bar{s}_{cmd}| + |\bar{s}_{err}| = n_I$, $|\bar{s}_{fix}| = n_O$, and n_L is the length of the longest string appearing in any of the three lists in the input example. To compute the set $\text{ALLSUBSTRINGS}(e, V, i)$ we iterate over all variable indices and for each variable index $j \in V$ we do the following.

- (1) Extract the string s_j corresponding to the variable $\text{VAR}(j) - O(n_I)$ iterations.
- (2) For each string s that is a substring of both $\bar{s}_{fix}[i]$ and s_j , compute all possible pairs of indices k_1, k_2 such that $\text{substr}(s_j, k_1, k_2) = s - O(n_K^2)$ possible substrings and $O(n_K)$ possible ways to place the substring in $\bar{s}_{fix}[i]$.
- (3) For each k_1 (resp. k_2) compute every position expression p_1 (resp. p_2) such that evaluating p_1 (resp. p_2) on s_j produces the index k_1 (resp. k_2) - $O(n_K)$ possible positions.
- (4) For each of these possibilities yield the expression $\text{SUB}(p_1, p_2, l, r, \text{VAR}(j))$ where l and r are such that $\bar{s}_{fix}[i] = l \cdot \text{substr}(s_j, k_1, k_2) \cdot r$.

ALLSUBSTRINGS produces a set of expressions that in the worst case has size $O(n_I n_K^5)$. If we restrict the offset component δ to only range over the values $\{-1, 0, 1\}$ for the symbolic expressions

$\text{CP}(c, i, \delta)$, the size reduces to $O(n_I n_K^3)$, and the synthesis algorithm is still sound and complete for this fragment of FIXIT .

This last restriction of the language can capture all the rules we are interested in. Notice that this analysis still holds in the extreme case in which all input matches are variable expressions of the form $\text{VAR-MATCH}(i, \epsilon, \epsilon)$. In our experiments on real-world commands, worst-case performance is uncommon, and is induced by substring operations over heterogeneous strings which yield many possible implementations. Consider the following two examples.

cmd1: aaaa aaaa	cmd2: bbbb bbbb
err1: aaaa aaaa	err2: bbbb bbbb
fix1: aa	fix1: bb

Performing synthesis on this pair of examples yields a pattern match consisting of four $\text{VAR-MATCH}(i, \epsilon, \epsilon)$ expressions. Due to the uniformity of the input strings, synthesis yields 48 possible SUB expressions. In particular, the desired fix can be generated from any of the four strings in the supplied s_{cmd} and s_{err} . Each of the four strings has three substrings of length 2, any of which yields the desired output. For each such substring, there are four pairs of IP values that supply the appropriate indices: The pair with two positive indices, the pair with two negative indices, and the two pairs consisting of one positive and one negative index.

Key point At this point we are ready to explain why all the match expressions can be kept as constants for as long as possible. If after processing a set of examples E , some expression in cmd or err is of the form $\text{STR}(s_i)$, then, for every input example, the value of the i -th component is the string s_i . Therefore, even if we replace this expression with a variable, all its instantiations will have the same values. Consequently, every function of the form $\text{SUB}(p_1, p_2, l, r, \text{VAR}(i))$ will produce a constant output, making it equivalent to the some constant function $\text{FSTR}(s')$.

Avoiding redundancy We discuss further improvements that make our symbolic rule representation more succinct. In the set of fix expressions enumerated by the function ALLSUBSTRINGS , the last three components of the expression $\text{SUB}(p_1, p_2, l, r, \text{VAR}(j))$ are often repeated many times. Looking at Figure 4d we can see how all the synthesized functions have $l = r = \epsilon$ and are applied to the variable $\text{VAR}(1)$. We define a data structure for representing sets of fix expressions that avoids these repetitions. A set of fix expressions is represented symbolically using a partial function $d : \mathbb{N} \mapsto (\Sigma^* \times \Sigma^*) \mapsto \text{Set}(P \times P)$ where P is the set of all position expressions. Formally, given a variable index i and two strings l and r , the set $d(i, l, r)$ symbolically represents the set of fix expressions $\{\text{SUB}(p_1, p_2, l, r, \text{VAR}(i)) \mid (p_1, p_2) \in d(i, l, r)\}$. The function d can be efficiently implemented and avoids redundancy. Considering again the example rule in Figure 4d, the fix expressions in the second component of the output can be succinctly represented by the function d that is only defined on the input $(1, \epsilon, \epsilon)$ such that

$$d(1, \epsilon, \epsilon) = \{(\text{IP}(0), \text{IP}(-5)), (\text{IP}(0), \text{CP}(\cdot, 1, 0)), (\text{IP}(0), \text{CP}(\cdot, -1, 0))\}.$$

4.3 Concrete Outputs

Taking into account the updated data structures, the algorithm SYNTRULES returns a symbolic rule r of the form $\text{match } cmd \text{ and } err \rightarrow_s \text{ fixes}$ where $cmd = [c_1, \dots, c_n]$ and

$err = [e_1, \dots, e_m]$ are lists of expressions of the form $STR(s)$ or $(VAR(i), B)$, while $fixes = [f_1, \dots, f_l]$ is a list of expressions of the form $FSTR(s)$ or (d, B) where d is the data structure for representing multiple fix expressions. The set of concrete FIXIT rules induced by this symbolic representation is the following.

$$\begin{aligned} \text{con}(\text{match } cmd \text{ and } err \rightarrow_s fix) &= \\ &\{\text{match } cmd \text{ and } err \rightarrow f \mid f \in \text{con}(fix)\} \\ \text{con}([f_1, \dots, f_l]) &= \{[f'_1, \dots, f'_l] \mid f'_i \in \text{con}(f_i)\} \\ \text{con}(FSTR(s)) &= \{FSTR(s)\} \\ \text{con}(d, B) &= \{\text{SUB}(p_1, p_2, l, r, VAR(i)) \mid \exists i, l, r. (p_1, p_2) \in d(i, l, r)\}. \end{aligned}$$

4.4 Synthesizing Multiple Rules

The algorithm we presented can synthesize all rules consistent with a set of examples in polynomial time. In practice, no single rule will be able to match all the examples gathered by the system. Instead, different examples will correspond to different rules—e.g., a rule for inserting `.java`, or a rule for moving directories. Here, we briefly describe how the algorithm presented in Figure 5 is used by NoFAQ to synthesize all such rules.

Given the set of examples E , NoFAQ maintains the set of all symbolic rules R consistent with some subset of the input examples and keeps track of which examples are consistent with which rules. In particular, for every subset $E' \subseteq E$, $\text{SYNTRULES}(E') = r$ and $r \neq \perp$, then $r \in R$. When a new example e is added, we iterate over all rules $r \in R$ and use the `REFINERULE` function to see whether each r can be modified to add e . If that's the case, the new rule is added to the system. When bad examples are flagged by the users, we remove all rules that contained the problematic example. In the worst case, there can be exponentially many subsets of E for which some consistent rule exists and our data structure can blow-up. As shown in Section 6, this happens infrequently in practice.

5 FORMAL PROPERTIES

We study the formal properties of the synthesis algorithm presented in Figure 5 and of the language FIXIT. First, our synthesis algorithm is invariant with respect to the order in which the training examples are presented. Thus, the properties of a symbolic rule generated by `SYNTRULES`, can be discussed solely in terms of the *set* of examples provided to `SYNTRULES`.

THEOREM 5.1 (ORDER INVARIANCE). *Given a list of examples E , for every permutation of examples E' of E , we have $\text{con}(\text{SYNTRULES}(E)) = \text{con}(\text{SYNTRULES}(E'))$.*

Second, the synthesis algorithm produces only rules that are consistent with the input examples. If we select an arbitrary concrete rule r from the set specified by a symbolic rule generated by `SYNTRULES`, and run it on the command and error of any of the examples provided to `SYNTRULES` for the synthesis of r , we will obtain the fix originally provided in that example.

THEOREM 5.2 (SOUNDNESS). *Given a list of examples E , for every rule $r \in \text{con}(\text{SYNTRULES}(E))$ and for every example $(\bar{s}_{cmd}, \bar{s}_{err}, \bar{s}_{fix}) \in E$, $\llbracket r \rrbracket(\bar{s}_{cmd}, \bar{s}_{err}) = \bar{s}_{fix}$.*

Since parts of the match expressions are “promoted” to variables only when the input examples show that this is required, our synthesis algorithm does not explicitly keep track of all the possible rules that can be consistent with the examples.

Our completeness result reflects this idea. Namely, the symbolic rule generated by an execution of `SYNTRULES` on a set of examples E implicitly encodes all FIXIT rules consistent with E in the following sense: either a rule r can be directly obtained via the concretization operator `con`, or there exists an example e which can be added to E to induce the creation of more symbolic variables. After adding e , r can be obtained by concretizing the symbolic rule.

THEOREM 5.3 (COMPLETENESS). *Given a non-empty set of examples E , for every FIXIT rule r that is consistent with E , either $r \in \text{con}(\text{SYNTRULES}(E))$ or there exists an example e such that $r \in \text{con}(\text{SYNTRULES}(e :: E))$.*

6 IMPLEMENTATION AND EVALUATION

We now describe the implementation details of NoFAQ, as well as our experimental evaluation of NoFAQ on a set of examples and test cases taken from `THEFXXX` and web forums.

6.1 Implementation

The NoFAQ web interface is implemented as a JavaScript and HTML website which uses AJAX calls to communicate with the synthesis server. We implemented the language FIXIT and its synthesis algorithm in F# together with the following additional optimizations. Since, for each symbolic rule, there can be multiple possible expressions in FIXIT that are consistent with the examples, we employ a simple and natural ranking technique to select an expression amongst them. If there are multiple `SUB` expressions that can generate the desired output string, we select the expression that uses the variable with the lowest index—i.e., the leftmost one. Similarly, the l and r included in `VAR-MATCH` expressions implicitly encode all rules matching on prefixes and suffixes of l and r , respectively. We choose the expression with the longest l and r .

6.2 Controlled Evaluation

We first assess the expressiveness of NoFAQ by evaluating it on a benchmark suite that includes the rules in the tool `THEFXXX`. We then evaluate its performance and scalability on additional benchmarks obtained from our web interface. The timing experiments were run on an Intel Core i7 4.00GHz CPU with 8 GB of RAM. We present both qualitative and quantitative analysis of the evaluation.

6.2.1 Qualitative Evaluation. In this section, we evaluate the expressiveness of FIXIT and the accuracy of our synthesis algorithms on a selected set of benchmarks. We compiled our benchmark suite from an initial set of 92 benchmarks, which were collected from both `THEFXXX` (76) and online help forums (16). Since rules in `THEFXXX` can use arbitrary Python code, it is hard to exactly compare them to the ones learned by NoFAQ. We use manual testing to check that a rule r generated by NoFAQ is consistent with a rule r' in `THEFXXX`. To do so, we manually constructed a set of examples based on the pattern-matching and textual substitutions performed by the `THEFXXX` rules. The other 16 example sets were obtained from examples found on command-line help forums on the web. These commands consist of various types of `git`, `svn`, and `mvn` commands, including committing, reverting, and deleting from repositories, as well as installing and removing packages. These 92 sets of examples

are also the ones we used to bootstrap an initial set of rules in our web interface. Cumulatively, we provided 168 examples.

For each repair rule, we observed that it was natural to provide 2 to 5 examples per benchmark for NoFAQ to uniquely learn the rule. We also provided additional examples for manually testing the learned rules, yielding a set of 3 to 8 examples. While these examples are synthetic examples reverse engineered from other sources, they are also natural examples, exercising the range of e.g. parameters, paths and file names one would expect to see in real Unix systems. In the case of the repaired command in Section 2.2, the natural two-example set would consist of two distinct directory names which do not share prefixes and suffixes, as well as filenames with distinct prefixes and extensions. For example, we used the two examples in Figure 4 and another example with the file name `Employee.java`.

Results We provided all collected examples to NoFAQ and our algorithm synthesized 108 rules, each consistent with some subset of the examples. In this count, if there was a rule r learned from a set of examples S and rule r' learned from a set $S' \subset S$, we only consider the rule r as it is more precise than the rule r' . We then performed a leave-one-out analysis to assess whether the learned rules generalized. For each example $e = (\bar{s}_{cmd}, \bar{s}_{err}, \bar{s}_{fix})$, we only considered the rules learned without using e and checked how many rules applied to the input $(\bar{s}_{cmd}, \bar{s}_{err})$ and whether one of those rules produced the correct output \bar{s}_{fix} .

For 69 of the 92 sets, regardless of what example e we left out of each set, only one of the learned rules matched the input of e and it always produced the correct output. For 12 out of 92 sets, regardless of what example e we left out of each set, multiple rules matched the input of e and at least one gave the fix in the example e . These are commands for which multiple fixes are possible and for which the system provides multiple suggestions. For 11 sets, no rule was learned that could generalize to the example left out. In summary, if we omit the constant rules that only matched a single example, NoFAQ could learn rules that collectively provided the intended suggestions for 81 of the example sets.

In some cases, for the same example set corresponding to a single THEFXXX rule, we had to synthesize more than one FIXIT rule to capture the different possible behaviors of a single rule in THEFXXX. For example, one can try adding 'sudo' in front of a command for several possible errors such as "Command not found", "You don't have the permission" etc. These are the cases in which more than 3 examples were given and NoFAQ generated a separate rule for each possible error message.

We now discuss the 11 THEFXXX rules that FIXIT could not learn to generalize. Two rules were checking complex properties of the input that FIXIT cannot capture. For example, FIXIT cannot check whether the error message contains some special character. FIXIT's conditional matching is limited to whole string or prefix/suffix matching, and thus cannot check if e.g. a file name contains a non-unicode whitespace character. FIXIT also cannot check whether some string in the input command is repeated more than once. Eight rules had hard-coded operations that were searching some context (the file system, a configuration file, etc.) for specific strings to complete the output. FIXIT only receives as inputs the command and the error message, and the rules currently cannot use any

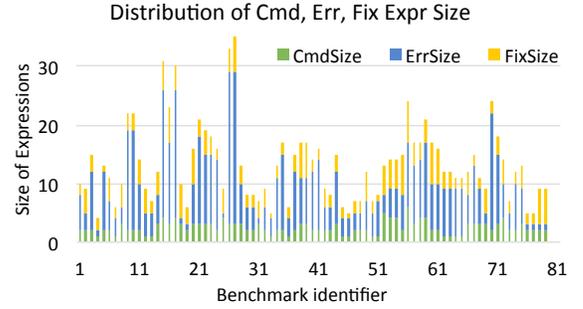


Figure 6: The distribution of benchmark sizes in terms of individual sizes of \bar{s}_{cmd} , \bar{s}_{err} , and \bar{s}_{fix} .

context drawn from e.g. file system or OS state. The remaining rule was both checking complex properties of the input and using context. Finding ways to use context is an interesting research direction, but there might be privacy issues for getting access to user's context such as the directory contents.

Distribution of rule sizes We define the size of an expression such as \bar{s}_{cmd} , \bar{s}_{err} , and \bar{s}_{fix} as the number of strings present in it. The distribution of the size of the benchmarks in terms of the sizes of the \bar{s}_{cmd} , \bar{s}_{err} , and \bar{s}_{fix} tuples in input-output examples is shown in Figure 6. Note that we do not show two benchmarks in the graph with disproportionately high \bar{s}_{err} expression size were learned from a single example. The average total size of the examples in the benchmarks was 15.91 ± 17.18^4 , with the maximum size of 116. The average sizes for the individual expressions of the examples were: i) \bar{s}_{cmd} : 2.38 ± 1.01 with maximum of 6, ii) \bar{s}_{err} : 10.12 ± 16.85 with a maximum of 110, and iii) \bar{s}_{fix} : 3.41 ± 1.55 with a maximum of 7. The output components of the synthesized rule contain on average $29.01\% \pm 24.4\%$ SUB expressions. Concretely, a synthesized rule contains on average 0.91 ± 0.76 SUB expressions.

Ranking Consistent with our hypothesis in Section 6.1, a diverse set of 2 to 3 examples was sufficient for eliminating spurious restrictions and substring expressions. In every test case, the rule chosen by our ranking was capable of correcting the held-out test cases. In practice, many rules still have several possible correct SUB expressions. However, this remaining ambiguity occurs because the same string can appear many times in the command and error message (e.g., the string `Employee` in the example in Section 2.1).

6.2.2 Quantitative Evaluation. We evaluate the performance of our synthesis algorithm with and without lazy VSA.

Evaluation of lazy VSA synthesis time In this experiment, we also consider the examples collected from our web interface. In Figure 7, we show the cumulative time taken to synthesize FIXIT rules for the 649 examples of repairs, using both the lazy and a non-lazy rule representation. This set contains the 168 examples we manually provided as well as 481 examples we obtained from our online tool. These examples yield a total of 1257 rules. The non-lazy representation always considers match and fix expressions as variables, rather than initially starting with constants. The lazy VSA

⁴We use $a \pm s$ to denote an average a with standard deviation s .

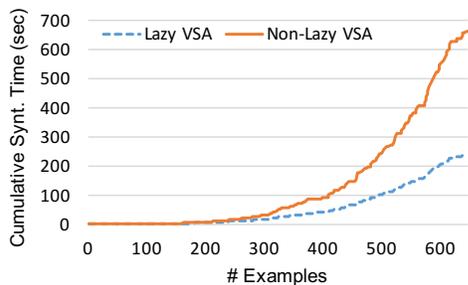


Figure 7: Cumulative synthesis time for n examples using the lazy and non-lazy rule representations.

takes an average of .37 seconds to process each individual example and took, in the worst case, 9.2 seconds to process an individual example. The non-lazy VSA takes an average of 1.0 seconds to process each individual example and took, in the worst case, 23.6 seconds to process an individual example. The widest difference between processing times for a single example was 16.9 seconds. Therefore, the non-lazy VSA incurs a significant overhead, and scales much worse than the lazy version. In summary, the lazy VSA strictly outperforms non-lazy VSA and can handle much larger sets of examples, a crucial requirement for our online application. Since we expect many more examples to be added to NoFAQ, a speed up of tens of seconds for synthesis-intensive examples is of critical importance, as maintaining high throughput ensures new rules are made visible as soon as possible.

Scalability of synthesis algorithm with example size Since all real-world examples we collected are relatively of small size (with maximum size of 116 space-separated strings), we evaluate the scalability of the SYNTHRULES algorithm by creating artificial examples of increasing sizes. We create these artificial examples by repeating the \bar{s}_{cmd} , \bar{s}_{err} , and \bar{s}_{fix} commands multiple times for the example shown in Section 2.1. The synthesis times are shown in Figure 8. From the graph, we observe that the synthesis times scale in a quadratic fashion with respect to the example size.

6.3 Live Deployment

We deployed a publicly available live version of NoFAQ. We received 1721 queries asking for ways to fix commands, and 496 contributed fixes to queries for which we did not have an answer. During this first deployment, a fraction of users was malicious and started posting meaningless queries as well as proposing “bad” fixes: one user proposed `rm -rf /` as a fix to many of the commands. We therefore introduced vetting mechanisms such as down-voting and blacklisting and these types of examples quickly disappeared.

Users are enthusiastic and have been contributing interesting examples. For example, the tool learned how to fix a command of the form “`find -name ‘test*.’`” by moving the dot before `-name`, as `find` expects the paths to search to come before other arguments, this is corrected with “`find . -name ‘test*’`”. To our surprise, the tool has also learned how to fix typos and generalize flag names in interesting ways. For example, after being presented with various misspellings like “`gti status`” or “`gitt status`”, the

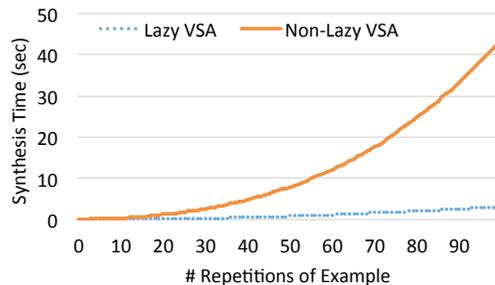


Figure 8: Synthesis times with increasing size of examples.

tool isolated a rule which maps commands that start with ‘g’ and are followed by ‘status’ to “`git status`”.

6.4 Threats to Validity

With respect to construct validity, in our controlled evaluation with manually provided examples, we have no formal way to check whether the rules learned by NoFAQ are semantically equivalent to those present in THEFXXX. We tried to attenuate this threat by manually inspecting the rules and using additional examples for testing the learned rules. For our live deployment experiment, we cannot guarantee that all the rules learned by NoFAQ are indeed meaningful as crowd-sourced examples can be noisy or incorrect. We mitigate this aspect by providing an additional interface that allows users to flag incorrect rules or examples.

Concerning internal validity, since we manually provided the examples to train the rules from THEFXXX, these examples may not be representative to the examples provided by a real user. To mitigate this threat, we provided NoFAQ additional set of examples and performed all combinations of leave-one-out cross-validation to evaluate whether any subset of the examples would yield the desired result on held-out examples.

Concerning external validity, since we do not know the identity and skills of our contributors, the corpus of crowd-sourced examples might not be representative of the kind of mistakes users make in the real world. Specifically, NoFAQ is capable of correcting mistakes for which enough of the necessary context can be extracted from the command and error message. Usage of StackOverflow, however indicates that users do in fact encounter such errors with common commands such as `java` [11], `find` [3], and `mkdir` [1]. In the case of the `java` command, users were invoking the Java runtime with the name of the source or class file, rather than the class to run; this problem was common enough that Oracle saw fit to include it in their official documentation [19]. We developed a web interface for NoFAQ to collect a large number of examples that will over time be more representative of the repairs needed by real users.

7 RELATED WORK

Version-space algebra (VSA) for synthesis The concept of VSA was first introduced by Mitchell [18] in the context of machine learning and was later used by Lau et al. to learn programs from demonstrations in SmartEdit [12]. It has since been used for many

PBE systems from various domains including syntactic string transformations in FlashFill [6], table transformations [8, 27], number transformations [26], text extraction from semi-structured text files in FlashExtract [13], and transformation of semi-structured spreadsheets to relational tables in FlashRelate [2]. Our synthesis algorithm also uses VSA to succinctly represent a large set of conforming expressions. However, in contrast to previous approaches that represent all conforming expressions concretely and refine via intersection, our synthesis algorithm maintains a lazy representation of rules and concretizes the choices only when needed. Moreover, NoFAQ's carefully designed DSL operators and VSA yield a polynomial time synthesis algorithm, unlike the exponential time algorithms of previous approaches.

In particular, it is illustrative to compare the FlashFill DSL with FIXIT. While, like FIXIT, FlashFill synthesizes string manipulations from input-output examples, specific performance properties make it less suitable for learning from large sample sets. FlashFill scales poorly as the error messages increase in length, which is a common occurrence for our benchmarks. Other limitations, namely the lack of an offset operator in position expressions and support for finite hard-coded regular expression tokens make FlashFill unsuitable for learning SUB expressions. Moreover, FlashFill's binary concatenation operator is represented using a DAG structure for a given example. The DAG is intersected to find consistent expressions for a set of examples, yielding an exponential time algorithm. FIXIT instead possesses unary string operations constrained to specific variable terms. The FIXIT language is disjoint from FlashFill DSL, and is expressive enough to enable practical command repairs while also admitting a polynomial time synthesis algorithm.

The concept of Lazy VSA is related to the *least general generalization* from inductive learning [21, 22] where constants are promoted to variables on a by need basis. Our approach is more general than [21, 22] since it also allows predicates over the matched variables (i.e., prefix and suffix matching).

Programming by Examples (PBE) PBE has been an active research area in the AI and HCI communities from a long time [16]. In addition to VSA-based data wrangling [7], PBE systems have recently been developed for various domains including interactive synthesis of parsers [15], synthesis of functional programs over algebraic data types [4, 20], program refactorings [23], data structure manipulations [28], and network policies [30]. NoFAQ also learns repair rules from few input-output examples of buggy and fixed commands, but both our problem domain of learning command repairs and the learning techniques of using lazy VSA are different from these PBE systems.

Program repair Research in automated program repair focuses on automatically changing incorrect programs to make them meet a desired specification [5]. The main challenge is efficiently searching the space of all programs to find one that behaves correctly. The most prominent search techniques are enumerative or data-driven. GenProg uses genetic programming in the hope of converging to a correct version [14]. Data-driven approaches use the large amount of code that is publicly available online to synthesize likely changes to the input program [24, 29]. Prophet [17] is a patch generation system that learns a probabilistic application-independent model of correct code from a set of successful human patches. Unlike these

techniques that learn a global model of code repair across different applications, our technique learns command-specific repairs by observing how expert users fix their buggy commands – i.e., from both the incorrect command the user started with (together with the error message) and the correct command she wrote as a fix.

Crowdsourced Repairs HelpMeOut is a social recommender system that helps novice users facing programming errors by showing examples of how other programmers have corrected similar errors [9]. Unlike NoFAQ, HelpMeOut only shows related examples, and cannot apply learned information to generate new repairs. Refazer learns syntactic program transformations from examples [25]. MistakeBrowser uses Refazer to learn ways to fix incorrect programs in introductory programming assignments [10]. Transformations are learned from examples of students' bug fixes and used to help subsequent students who encounter mistakes that are similar to those seen in the past. When compared to NoFAQ, MistakeBrowser does not have strong theoretical guarantees and targets a different domain with different challenges. In particular, Refazer cannot learn string manipulations, only AST transformations.

THEFXXX provides a Python interface for writing substitution and repair rules, requiring skills not necessarily possessed by novice CLI users, particularly if a correction requires non-trivial string operations. Like FlashFill, we aim to emulate the workflow of non-technical users communicating with experts on web forums. For a command line novice, Python string manipulations may be challenging, and an incorrectly transformed command can be catastrophic. In a situation where a non-expert desires a new THEFXXX rule, they may provide an example of several command/error pairs, and the fix for each, from which an expert would write the desired Python code. NoFAQ shortens this loop by moving the fix synthesis into a polynomial time algorithm on the user's machine.

8 CONCLUSION AND FUTURE DIRECTIONS

We presented a tool NoFAQ that suggests possible fixes to common error-triggering commands by learning from examples of how experts fix such issues. Our language design walks a fine line between expressiveness and performance: by careful choice of unary operators over pre-defined variables, and exclusion of arbitrary substring operations, we avoid exponential-time worst case performance, while still maintaining a useful degree of functionality. NoFAQ was able to instantly synthesize 85% of the rules appearing in the popular repair tool THEFXXX and 16 other rules from online help forums. Moreover, our web version of NoFAQ is constantly receiving new queries and example repairs from users around the world. Although NoFAQ is aimed towards repairing commands, we believe our novel combination of synthesis and rule-based program repair is quite general and is applicable in many other domains as well. We plan to apply this methodology to more complex tasks, such as correcting syntax errors in source code, applying code optimizations, and editing configuration files.

ACKNOWLEDGEMENTS

Loris D'Antoni and Michael Vaughn were supported by a Google Faculty Research Award.

REFERENCES

- [1] baltostar (<https://unix.stackexchange.com/users/42571/baltostar>). How to create nested directory in a single command [duplicate]. StackOverflow. URL <https://unix.stackexchange.com/q/84191>. URL:<https://unix.stackexchange.com/q/84191> (version: 2014-11-07).
- [2] D. W. Barowy, S. Gulwani, T. Hart, and B. Zorn. Flashrelate: Extracting relational data from semi-structured spreadsheets using examples. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2015, pages 218–228, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3468-6. doi: 10.1145/2737924.2737952. URL <http://doi.acm.org/10.1145/2737924.2737952>.
- [3] cfinley (<https://stackoverflow.com/users/425683/cfinley>). ffind: paths must precede expression:fi How do I specify a recursive search that also finds files in the current directory? StackOverflow. URL <https://stackoverflow.com/q/6495501>. URL:<https://stackoverflow.com/q/6495501> (version: 2011-06-27).
- [4] J. K. Feser, S. Chaudhuri, and I. Dillig. Synthesizing data structure transformations from input-output examples. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*, pages 229–239, 2015. doi: 10.1145/2737924.2737977. URL <http://doi.acm.org/10.1145/2737924.2737977>.
- [5] C. Goues, S. Forrest, and W. Weimer. Current challenges in automatic software repair. *Software Quality Journal*, 21(3):421–443, Sept. 2013. ISSN 0963-9314. doi: 10.1007/s11219-013-9208-0. URL <http://dx.doi.org/10.1007/s11219-013-9208-0>.
- [6] S. Gulwani. Automating string processing in spreadsheets using input-output examples. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '11, pages 317–330, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0490-0. doi: 10.1145/1926385.1926423. URL <http://doi.acm.org/10.1145/1926385.1926423>.
- [7] S. Gulwani, W. R. Harris, and R. Singh. Spreadsheet data manipulation using examples. *Commun. ACM*, 55(8):97–105, Aug. 2012. ISSN 0001-0782. doi: 10.1145/2240236.2240260. URL <http://doi.acm.org/10.1145/2240236.2240260>.
- [8] W. R. Harris and S. Gulwani. Spreadsheet table transformations from examples. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011*, pages 317–328, 2011. doi: 10.1145/1993498.1993536. URL <http://doi.acm.org/10.1145/1993498.1993536>.
- [9] B. Hartmann, D. MacDougall, J. Brandt, and S. R. Klemmer. What would other programmers do: Suggesting solutions to error messages. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '10, pages 1019–1028, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-929-9. doi: 10.1145/1753326.1753478. URL <http://doi.acm.org/10.1145/1753326.1753478>.
- [10] A. Head, E. Glassman, G. Soares, R. Suzuki, L. D'Antoni, and B. Hartmann. Writing Reusable Code Feedback at Scale with Mixed-Initiative Program Synthesis. In *L@S'17: 4th ACM Conference on Learning at Scale*, 2017.
- [11] S. C. (https://stackoverflow.com/users/139985/stephen_c). What does fiCould not find or load main classfi mean? StackOverflow. URL <https://stackoverflow.com/q/18093928>. URL:<https://stackoverflow.com/q/18093928> (version: 2014-5-18).
- [12] T. Lau, S. A. Wolfman, P. Domingos, and D. S. Weld. Programming by demonstration using version space algebra. *Mach. Learn.*, 53(1-2):111–156, Oct. 2003. ISSN 0885-6125. doi: 10.1023/A:1025671410623. URL <http://dx.doi.org/10.1023/A:1025671410623>.
- [13] V. Le and S. Gulwani. Flashextract: A framework for data extraction by examples. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, pages 542–553, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2784-8. doi: 10.1145/2594291.2594333. URL <http://doi.acm.org/10.1145/2594291.2594333>.
- [14] C. Le Goues, M. Dewey-Vogt, S. Forrest, and W. Weimer. A systematic study of automated program repair: Fixing 55 out of 105 bugs for \$8 each. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 3–13, Piscataway, NJ, USA, 2012. IEEE Press. ISBN 978-1-4673-1067-3. URL <http://dl.acm.org/citation.cfm?id=2337223.2337225>.
- [15] A. Leung, J. Sarracino, and S. Lerner. Interactive parser synthesis by example. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*, pages 565–574, 2015. doi: 10.1145/2737924.2738002. URL <http://doi.acm.org/10.1145/2737924.2738002>.
- [16] H. Lieberman. *Your wish is my command: Programming by example*. Morgan Kaufmann, 2001.
- [17] F. Long and M. Rinard. Automatic patch generation by learning correct code. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL 2016, pages 298–312, 2016.
- [18] T. M. Mitchell. Generalization as search. *Artif. Intell.*, 18(2), 1982.
- [19] Oracle. Common Problems (and Their Solutions). Oracle Java Tutorials. URL <https://docs.oracle.com/javase/tutorial/getStarted/problems/>.
- [20] P. Osera and S. Zdancewic. Type-and-example-directed program synthesis. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*, pages 619–630, 2015. doi: 10.1145/2737924.2738007. URL <http://doi.acm.org/10.1145/2737924.2738007>.
- [21] G. D. Plotkin. A note on inductive generalization. *Machine Intelligence*, 5:153–163, 1970.
- [22] G. D. Plotkin. A further note on inductive generalization. *Machine Intelligence*, 6:101–124, 1979.
- [23] V. Raychev, M. Schäfer, M. Sridharan, and M. T. Vechev. Refactoring with synthesis. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2013, part of SPLASH 2013, Indianapolis, IN, USA, October 26-31, 2013*, pages 339–354, 2013. doi: 10.1145/2509136.2509544. URL <http://doi.acm.org/10.1145/2509136.2509544>.
- [24] V. Raychev, M. Vechev, and E. Yahav. Code completion with statistical language models. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14*, pages 419–428, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2784-8. doi: 10.1145/2594291.2594321. URL <http://doi.acm.org/10.1145/2594291.2594321>.
- [25] R. Rolim, G. Soares, L. D'Antoni, O. Polozov, S. Gulwani, R. Gheyi, R. Suzuki, and B. Hartmann. Learning syntactic program transformations from examples. In *Proceedings of ICSE*. IEEE Press, 2017.
- [26] R. Singh and S. Gulwani. Synthesizing number transformations from input-output examples. In *Proceedings of the 24th International Conference on Computer Aided Verification, CAV'12*, pages 634–651, Berlin, Heidelberg, 2012. Springer-Verlag. ISBN 978-3-642-31423-0. doi: 10.1007/978-3-642-31424-7_44. URL http://dx.doi.org/10.1007/978-3-642-31424-7_44.
- [27] R. Singh and S. Gulwani. Learning semantic string transformations from examples. *Proc. VLDB Endow.*, 5(8):740–751, Apr. 2012. ISSN 2150-8097. doi: 10.14778/2212351.2212356. URL <http://dx.doi.org/10.14778/2212351.2212356>.
- [28] R. Singh and A. Solar-Lezama. Synthesizing data structure manipulations from storyboards. In *SIGSOFT FSE*, pages 289–299, 2011.
- [29] M. Yakout, A. K. Elmagarmid, J. Neville, M. Ouzzani, and I. F. Ilyas. Guided data repair. *Proc. VLDB Endow.*, 4(5):279–289, Feb. 2011. ISSN 2150-8097. doi: 10.14778/1952376.1952378. URL <http://dx.doi.org/10.14778/1952376.1952378>.
- [30] Y. Yuan, R. Alur, and B. T. Loo. Netegg: Programming network policies by examples. In *Proceedings of the 13th ACM Workshop on Hot Topics in Networks, HotNets-XIII, Los Angeles, CA, USA, October 27-28, 2014*, pages 20:1–20:7, 2014. doi: 10.1145/2670518.2673879. URL <http://doi.acm.org/10.1145/2670518.2673879>.