Automatic Repair of Regular Expressions

RONG PAN, The University of Texas at Austin, USA
QINHEPING HU, University of Wisconsin-Madison, USA
GAOWEI XU, University of Wisconsin-Madison, USA
LORIS D'ANTONI, University of Wisconsin-Madison, USA

We introduce RFixer, a tool for repairing complex regular expressions using examples. Given an incorrect regular expression and sets of positive and negative examples, RFixer synthesizes the closest regular expression to the original one that is consistent with the examples. Automatically repairing regular expressions requires exploring a large search space because practical regular expressions: *i)* are large, *ii)* operate over very large alphabets—e.g., UTF-16 and ASCII—and *iii)* employ complex constructs—e.g., character classes and numerical quantifiers. RFixer's repair algorithm achieves scalability by taking advantage of structural properties of regular expressions to effectively prune the search space, and it employs satisfiability modulo theory solvers to efficiently and symbolically explore the sets of possible character classes and numerical quantifiers. RFixer could successfully compute minimal repairs for regular expressions collected from a variety of sources, whereas existing tools either failed to produce any repair or produced overly complex repairs.

CCS Concepts: • **Software and its engineering** → *General programming languages*; • **Theory of computation** → **Regular languages**.

Additional Key Words and Phrases: Program Repair, Regular Expressions, Program Synthesis

ACM Reference Format:
Rong Pan, Qinheping Hu, Gaowei Xu, and Loris D'Antoni. 2019. Automatic Repair of Regular Expressions. *Proc. ACM Program. Lang.* 3, OOPSLA, Article 139 (October 2019), 29 pages. https://doi.org/10.1145/3360565

1 INTRODUCTION

One of the most profound impacts of computing has been to enable a broad range of disciplines—from linguistics to geology—to collect, store and analyze an ever expanding set of data about the phenomena they study [Hey et al. 2009]. Similarly, jobs from advertising to machine maintenance are being transformed into data-centric occupations. As a result, providing support for filtering, transforming, and analyzing data has become crucial. In this paper, we focus on helping programmers write regular expressions, which not only are the de-facto tool for extracting data from unstructured datasets, but are also part of the standardized computer science curriculum and are taught in data science and theory of computation courses [url 2018].

We introduce RFixer, a tool for helping programmers repair regular expressions. We only consider regular expressions that do not contain non-regular operators—e.g., negative lookahead. Although we do not consider non-regular operators, our tool is expressive enough to capture most of the

Authors' addresses: Rong Pan, The University of Texas at Austin, USA, rpan@cs.utexas.edu; Qinheping Hu, Department of Computer Sciences, University of Wisconsin-Madison, 1210 West Dayton Street, Madison, WI, 53706, USA; Gaowei Xu, Department of Computer Sciences, University of Wisconsin-Madison, 1210 West Dayton Street, Madison, WI, 53706, USA; Loris D'Antoni, Department of Computer Sciences, University of Wisconsin-Madison, 1210 West Dayton Street, Madison, WI, 53706, USA.

This work is licensed under a Creative Commons Attribution 4.0 International License.
© 2019 Copyright held by the owner/author(s).
2475-1421/2019/10-ART139
https://doi.org/10.1145/3360565

regular expressions appearing in practical applications.[1] RFixer takes as input a regular expression and a set of positive and negative examples—i.e., strings the regular expression should respectively accept and reject. When the regular expression is incorrect on the examples, RFixer automatically synthesizes the syntactically smallest repair of the original regular expression that is correct on the given examples.

The problem tackled by RFixer is challenging because practical regular expressions *i)* are large, *ii)* operate on very large alphabets consisting of upwards of hundreds of characters, and *iii)* employ complex constructs such as character classes (e.g., `[a-z0-9]`) and numerical quantifiers (e.g., `\d{8,15}`, which describes a sequence of digits of length between 8 and 15 characters). The main contribution of this paper is the first sound and complete algorithm for finding minimal repairs that can scale to practical regular expressions. Our algorithm achieves scalability by taking advantage of structural properties of regular expressions to localize what sub-expressions should be modified and employs satisfiability modulo theory (SMT) solvers to efficiently and symbolically explore the set of possible character classes and numerical quantifiers.

Algorithm. At the high level, RFixer operates as follows. Starting from the initial regular expression—e.g., `[a-z]{3,5}[0-9]*`—and positive and negative examples—e.g., $P = \{$abc4, ac4$\}$ and $N = \{$a12$\}$—RFixer generates a set of initial templates—i.e., regular expressions with holes. On our example a possible template is `○{▹,◃}[0-9]*` where ○ is a hole that can be replaced by any regular expression and ▹ and ◃ are quantifier holes that can be replaced by numbers. RFixer then processes the templates in order of distance from the original expression and, for each template t, performs the following three steps.

Template pruning. RFixer performs a polynomial-time test to check whether the template t can ever result in a correct repair, otherwise it discards it. In our example, the template `○{▹,◃}[0-9]*` is kept because `.*[0-9]*` (`.` is the character class containing every character) accepts all positive examples and `∅[0-9]*` rejects all negative examples. However, the template `[a-z]{3,5}○` is discarded because no instantiation of it can accept the positive example ac4.

Simple completion. If the template is not discarded, RFixer tries to find a correct instantiation of the template that replaces each hole of the form ○ with a set of characters. We show that this problem is NP-Hard and propose two SMT encodings for solving it. Our encodings are based on the automata and declarative semantics of regular expressions and have different complexities and orthogonal performance.

Template generation. Last, our algorithm adds to the set of unexplored templates with new templates obtained by generating new holes in the current template using regular expression operators—e.g., expand `○{▹,◃}[0-9]*` to `(○|○){▹,◃}[0-9]*`.

Evaluation. RFixer could produce minimal repairs that were consistent with the given examples for 1,588/2,104 regular expressions with small alphabets from the personalized education website Automata Tutor [Tutor 2015] (number of examples varying between 4 and 96 per expression), 23/25 real regular expressions collected from the regular expression forum RegExLib [RegExLib 2017] (number of examples varying between 10 and 39 per expression), and 35/50 regular expressions from Rebele et al. [2018] (number of examples varying between 17 and 75 per expression) with an average time of 16.4 seconds. Most important, RFixer produced high-quality repairs. First, using a counterexample-guided inductive synthesis algorithm RFixer could also produce semantically correct (w.r.t. a specification) repairs for 1,156/2,104 Automata Tutor expressions. Second, the

[1]We analyzed a large public collection of regular expressions (https://github.com/lorisdanto/automatark/tree/master/regex) from a variety applications—e.g., string matching and deep-packet inspection—and only 724/6124 expressions used non-regular operators.

repairs produced by RFixer on the problems from Rebele et al. [2018] generalized well to left-out example sets. In particular, RFixer's F1 scores on left-out examples were on average 56% higher than the F1 scores of the state-of-the-art tool from Rebele et al. [2018]!

Intended Use of RFixer. Our preliminary results show that RFixer could be used in a variety of applications. First, RFixer can be deployed in tools like Automata Tutor [D'Antoni et al. 2015a] and used to build feedback systems for helping students understand regular expressions. Second, RFixer can be deployed as a helping tool for debugging regular expressions in systems like https://regexr.com/, which provide intuitive interfaces for people to experiment with regular expressions. Even though in this scenario a full specification of the intended behavior of the regular expression is not available, RFixer often produces high-quality repairs that have correct templates and generalize to unseen data. In the rare cases where RFixer produces low-quality repairs, we believe programmers can benefit from seeing suggestions on how to fix individual examples and use the suggestions to generalize the fixes provided by RFixer.

Contributions. In summary, our contributions are:

- RFixer: the *first sound and complete tool* for producing minimal repairs from examples for regular expressions with complex alphabets, character classes, and numerical quantifiers (§ 2).
- A synthesis algorithm that uses structural properties of regular expressions to efficiently prune the search space of regular-expression templates and uses SMT solvers to avoid explicitly searching the large space of character classes and numerical quantifiers (§ 4 and 5). We propose two SMT encodings with orthogonal performance for solving individual templates. The encodings are based on different semantics of regular expressions: one based on automata and one based on regular expression operators.
- A comprehensive evaluation of RFixer on three representative sets of benchmarks (§ 6). First, for 2,104 regular expressions over small alphabets submitted by students as homework assignments in the tool Automata Tutor [Tutor 2015], RFixer successfully produced 1,588/2,104 repairs consistent with the given examples and 1,156/2,104 repairs equivalent to the correct assignment solutions. Second, for 25 regular expressions over the ASCII alphabet from the popular regular expressions website http://www.regexlib.com, RFixer successfully repaired 23/25. Third, for 50 regular expression over the ASCII alphabets from Rebele et al. [2018], RFixer successfully repaired 35/50 expressions and produced repairs that on average had 56% higher accuracy on left-out examples than those produced by the state-of-the-art tool from Rebele et al. [2018].

2 ILLUSTRATIVE EXAMPLE

We illustrate the main ideas behind RFixer using a concrete example of an incorrect regular expression collected from the regular expression website RegExLib [RegExLib 2017]. A user has shared on the website the following regular expression that they claim correctly accepts valid MasterCard™ numbers starting with the numbers 51 through 55.

```
[51|52|53|54|55]{2}[0-9]{14}
```

However, the user has misunderstood the syntax of regular expressions and made a few mistakes that cause the expression to misbehave on several inputs. In particular, when the pipe character | is used within square brackets, it does not denote the union operator but the character '|' itself (i.e., the expression is equivalent to `[12345|]{2}[0-9]{14}`). After posting the incorrect regular expression on the website, several users pointed out that the regular expression was incorrect and

```
5125632154125412    5225632154125412    5525632154125412    5525632154125412
512563215412541 2   5325632154125412    5425632154125412    5425632154125412
                            5211632154125412
```

(a) Positive examples

```
1525632154125412    2525632154125412    3525632154125412    1599999999999999
4525632154125412    |525632154125412    3525632154125412    5625632154125412
4825632154125412   6011632154125412   6011-1111-1111-1111  5423-1111-1111-1111
                            341111111111111
```

(b) Negative examples

```
[51|52|53|54|55]{2}[0-9]{14}                    5[12345][0-9]{14}
```

(c) Incorrect expression (d) Repaired expression produced by RFixer

Fig. 1. Example of repair produced by RFixer.

provided several counterexamples. For example, a user mentioned that strings starting with `15` or `|5` are accepted even though they shouldn't be.

Using these counterexamples, a user can ask RFixer to automatically repair the incorrect expression (Figure 1). When interacting with RFixer, a user provides a regular expression together with positive and negative examples. The user then asks RFixer to repair the original expression and RFixer computes the regular expression that correctly produces the matches and is "closest" to the original one. In this example, RFixer takes 16 seconds to output the following expression, which correctly fixes the user's mistakes.

```
5[12345][0-9]{14}
```

RFixer computes the repaired expression as follows. First, RFixer iteratively transforms the original expression into templates—i.e., regular expressions with holes. Each template implicitly represents all possible regular expressions that can be obtained by instantiating the values of the holes. Enumerating templates instead of complete regular expressions addresses the problem of explicitly enumerating complex regular expression features such as character classes and numerical quantifiers. For example, the template `∘([0-9]{14})` is obtained by replacing the first character class and its quantifier in the original expression with a hole. RFixer explores templates in order by preferring templates that are closest to the original expression first. For every template, RFixer performs a series of tests to decide whether the template will ever result in a correct regular expression based on what kind of values the hole can assume. For example, the template `∘([0-9]{14})` cannot result into a correct regular expression if we only replace the hole $\circ$ with a character class. RFixer reaches this conclusion by observing that the regular expression `.([0-9]{14})`—in which the hole has been replaced with the universal character class `.`—does not match all positive matches. RFixer performs several other tests to further prune the search space of possible templates.

Second, RFixer needs to find concrete values to replace the holes with and produce a complete regular expression. In the example, we assume that RFixer is now processing the template

$$(\circ_1\circ_2)\{\triangleright,\triangleleft\}([0\text{-}9])\{14\}$$

In this stage, RFixer assumes that $\circ_1$ and $\circ_2$ can only be replaced with character classes, and $(\triangleright,\triangleleft) \in \mathbb{N} \times (\mathbb{N} \cup \{\infty\})$. RFixer generates an SMT formula to find what characters should belong to the character classes that should instantiate the holes $\circ_1$ and $\circ_2$, and what numbers should belong to $\mathbb{N} \times (\mathbb{N} \cup \{\infty\})$ to instantiate the holes $\triangleright$ and $\triangleleft$. For example, the following constraint is generated to denote that the final result should accept the string `5125632154125412`: $v_1^5 \wedge v_2^1 \wedge \triangleright{\leq}1 \wedge \triangleleft{\geq}1$. Here, v_1^5 is set to true to denote that the character class for hole $\circ_1$ should contain the character 5, and $\triangleright \leq 1$ denotes $(\circ_1\circ_2)$ should repeat at most once. RFixer solves the generated constraint and returns the correct expression mentioned above. On the same example, the synthesis tool RegexGenerator++ [Bartoli et al. 2016] takes more than 100 seconds to produce the following incorrect regular expression that is completely different from the original one:

```
\w[123]\14|\w427654851354895|[555][555]\w\w++
```

3 THE REPAIR PROBLEM

In this section, we formally define the regular expression repair problem.

Preliminaries. We assume a finite totally ordered alphabet Σ and use $a, b, \ldots$ to denote characters in Σ. We use $a_1{<}a_2$ to denote that character a_1 appears before a_2 in the total order. In our applications, Σ will be a common character set—e.g., ASCII, UTF-16, or $\{0, 1\}$. A string is a finite sequence of characters $s = a_1 \ldots a_n \in \Sigma^*$. The length of s is denoted as $|s| = n$. A language is a set of strings $L \subseteq \Sigma^*$.

Regular expressions. A regular expression is an expression formed using the operators

$$[C] \qquad r|r \qquad rr \qquad r\{m, M\}$$

where $C \subseteq \Sigma$ is a set of characters, $m \in \mathbb{N}$, and $M \in \mathbb{N} \cup \{\infty\}$. We use R to denote the set of all regular expressions. The language $L(r) \subseteq \Sigma^*$ of a regular expression r is inductively defined as $L([C]) = C$, $L(r_1|r_2) = L(r_1) \cup L(r_1)$, $L(r_1r_2) = \{s_1s_2 \mid s_1 \in L(r_1) \wedge s_2 \in L(r_2)\}$, and $L(r\{m, M\}) = \{s_1 \ldots s_n \mid m \leq n \leq M \wedge \forall i.\ s_i \in L(r)\}$. We can define common regular operators as follows: *i)* $\emptyset = [\emptyset]$, *ii)* $a = [\{a\}]$, *iii)* $r? \equiv r\{0, 1\}$, *iv)* $r^* \equiv r\{0, \infty\}$, *v)* $r^+ \equiv r\{1, \infty\}$, *vi)* $r\{i\} \equiv r\{i, i\}$ where $i \in \mathbb{N}$, and *vii)* $\varepsilon \equiv \emptyset\{0, 0\}$.

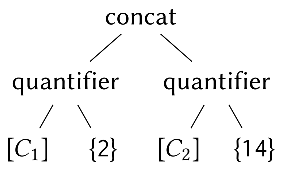
The operators $?, +, *, \{m, M\}$ and $\{i\}$ are often called *quantifiers.* Practical regular expressions support special operators called character classes to denote certain sets of characters. Common character classes are intervals $[c_1{-}c_2]$ denoting the set of characters $\{a \mid c_1{\leq}a{\leq}c_2\}$ and alphabet specific classes such as `\d`, which contains digits, and `\s`, which contains all space characters. In the following, we assume that every expression of the form $r\{m, M\}$ with $m > 0$ is such that $\varepsilon \notin L(r)$. When $\varepsilon \in L(r)$, we can always rewrite $r\{m, M\}$ as $r\{0, M\}$.

Distance between expressions. In this paper, we are interested in repairing regular expressions and we will prefer solutions that are "close" to the initial expression. Given an abstract syntax tree τ, an *edit* $\tau[\tau_1'/\tau_1, \ldots, \tau_n'/\tau_n]$ replaces each subtrees τ_i of the AST τ with a new subtree τ_i'. The cost of an edit is the sum of the number of nodes in every τ_i and τ_i'.[2]

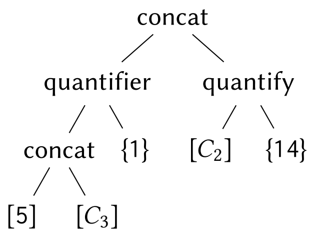
Definition 3.1. The *distance* $D(r_1, r_2)$ between two regular expressions r_1 and r_2 is the minimum cost of an edit that transforms r_1 into r_2.

Example 3.2. Figure 2 shows the syntax trees τ_1 and τ_2 for the expressions r_1 = `[51|52|53|54|55]{2}[0-9]{14}` and r_2 = `(5[12345]){1}[0-9]{14}` from Section 2. The tree τ_2 can be obtained by replacing the subtree $\tau_1^s = [C_1]$ in τ_1 with a new subtree $\tau_1'^s = \text{concat}([5], [C_3])$ and replacing the subtree $\tau_2^s = \{2\}$ by $\tau_2'^s = \{1\}$. Since τ_1^s, $\tau_1'^s$, τ_2^s and $\tau_2'^s$ have 1, 3, 1 and 1 nodes respectively, the distance $D(r_1, r_2)$ between the two expressions is $1 + 3 + 1 + 1 = 6$. Consider

[2]We choose a distance that captures the notion of replacing sub-expressions. Using a tree edit distance [Bille 2005] would not work well in this setting—e.g., the expressions `(ab|cd)` and `(abcd)` would have distance 1 from each other since one can simply replace a union node with a concatenation node.



AST τ_1 of the expression r_1.



AST τ_2 of the expression r_2.

Fig. 2. Syntax trees of the original and repaired expression from Figure 1. Here, C_1 = [51|52|53|54|55], C_2 = [0-9], and C_3 = [12345]. In Figure 1, we omitted the quantifier {1} appearing in τ_2 for clarity.

now r_3 = (5[12345]){1}[0-9]{1}. This expression has distance $D(r_1, r_3) = 6 + 1 + 1 = 8$ from r_1 because to obtain r_3 from r_1, we also need to replace {14} with {1}. □

Problem Statement. We are now ready to define our repair problem.

Definition 3.3 (Repair from examples). Given a finite set of positive examples $P \subseteq \Sigma^*$, a finite set of negative examples $N \subseteq \Sigma^*$ such that $P \cap N = \emptyset$, and a regular expression r, the *repair-from-examples problem* is to find a regular expression r' that is consistent with examples—accepts all positive examples (i.e., $P \subseteq L(r')$) and rejects all negative examples (i.e., $N \cap L(r') = \emptyset$)—and has minimal distance from r—i.e., there exist no r'' consistent with the examples such that $D(r, r'') < D(r, r')$.

Notice that the solution to the repair problem might not be unique. Moreover, if we set r to any expression of size 1 to start with, the repair problem becomes that of synthesizing the smallest regular expression consistent with a set of examples, which is an NP-hard problem [Gold 1978].

Theorem 3.4. *The repair-from-examples problem is NP-hard.*

Proof. Gold showed that synthesizing the smallest regular expression consistent with a set of examples is an NP-Complete problem [Gold 1978]. This problem trivially reduces to the repair-from-examples problem in which the initial regular expression is the expression $r = \varepsilon$. □

4 REPAIR ALGORITHM

We now present our algorithm for solving the repair-from-examples problem. Our algorithm enumerates regular-expression templates with holes of increasing distance from the original expression and tries to find a "simple" completion of such templates that is a solution to the repair problem. We first define templates and then describe the structure of our enumeration algorithm.

Definition 4.1 (Template). A *template* is an expression in the grammar

$$t := [C] \mid t|t \mid tt \mid t\{m, M\} \mid t\{\triangleright, \triangleleft\} \mid \circ$$

where $\circ$ is a *hole* (resp. $(\triangleright, \triangleleft)$ is a quantifier hole) from a set of holes $H_\circ$, (resp. a set of quantifier holes $H_{\triangleright\triangleleft}$).

Holes of the form $\circ$ always appear as a leaf in expressions and can be replaced with any regular expression while quantifier holes $\triangleright, \triangleleft$ can be replaced with any two possible values for the m, M quantifiers. Given a template t with holes $\circ_1, \ldots, \circ_n, (\triangleright_1, \triangleleft_1), \ldots, (\triangleright_k, \triangleleft_k)$, we write $t\langle r_1, \ldots, r_n, (m_1, M_1), \ldots, (m_k, M_k)\rangle$, to denote the result of replacing each hole with concrete $r_i \in R$ and $(m_j, M_j) \in \mathbb{N} \times (\mathbb{N} \cup \{\infty\})$. Intuitively, a template denotes an infinite set of possible regular expressions; formally the set of induced concrete regular expressions of a template t

Algorithm 1: RFixer search algorithm

```
/* r: regular expression, P positive examples, N negative examples */
function RegExRepair($r, P, N$)
    $Q$ ←getInitialTemplates($r$)
    while $true$ do
        $t \leftarrow Q$.pop()
        if $L(t_\bot) \cap N = \emptyset$ and $P \subseteq L(t_\top)$ then
            if $P \subseteq L(t_\Sigma)$ then
                $\Phi$ ←GenerateConstraint($t, P, N$)
                if Solve($\Phi$)=$SAT$ then
                    return solutionOf ($\phi$,t)
            $Q$.push(expandHoles($t$))
        $Q$.push(addOrReduceHoles($t$))
```

with holes $\circ_1, \dots, \circ_n, (\triangleright_1, \triangleleft_1), \dots, (\triangleright_k, \triangleleft_k)$ is $\textsc{con}(t) = \{t\langle r_1, \dots, r_n, (m_1, M_1), \dots, (m_k, M)\rangle \mid r_i \in R, m_j \in \mathbb{N}, M_j \in \mathbb{N} \cup \{\infty\}\}$.

Example 4.2. Consider the templates t_1=∘(`[0-9]{14}`) and t_2=$\circ_1\circ_2$(`[0-9]{14}`) from Section 2. The two regular expressions in Figure 2 are actually two possible completions of t_1: `[51|52|53|54|55][0-9]{14}` = $t_1\langle$`[51|52|53|54|55]`$\rangle$, and t_2: `[5][12345][0-9]{14}` = $t_2\langle$`[5][12345]`$\rangle$. □

The notion of distance from Definition 3.1 naturally lifts to templates and their abstract syntax trees. Our template search algorithm is shown in Algorithm 1. The algorithm maintains a queue Q of templates sorted by cost. The initial set of templates (line 2) is obtained by either replacing a character class $[C]$ in r with a hole $\circ$, or replacing a quantifier $\{m, M\}$ with $\{\triangleright, \triangleleft\}$.

In the main loop, the algorithm extracts a template t from the queue and performs the following three steps:

1. **Template pruning** Check if the template t can ever result in a correct repair (line 5, Sec. 4.1).
2. **Simple completion** Check if there exists a solution in which holes of the form $\circ$ are replaced with sets of characters $[C]$ (lines 6-9, Sec. 4.2).
3. **Template generation** Generate new templates using regular expression operators (line 10,11, Sec. 4.3).

4.1 When to Discard a Template

We propose a simple technique for efficiently discarding templates that are *guaranteed to not result into a solution* (Step 1 of the algorithm). Our technique is based on the fact that regular expressions only use monotonic operators[3]—i.e., given a template t with a hole $\circ$, and two regular expressions r_1 and r_2, the following holds:

$$\text{If } L(r_1) \subseteq L(r_2) \text{ then } L(t\langle r_1\rangle) \subseteq L(t\langle r_2\rangle).$$

We generalize this idea in the following theorem.

[3] Regular expressions support negation at the character class level, but not at the expression level. We still consider this form of negation to be monotonic, since character classes using negation can easily be rewritten to remove negation.

Theorem 4.3. *Given a template t with holes $\circ_1, \dots, \circ_n, (\triangleright_1, \triangleleft_1), \dots, (\triangleright_k, \triangleleft_k)$, let*

$$t_\bot = t\langle \emptyset, \dots, \emptyset, (1,0), \dots, (1,0)\rangle \quad \textit{and} \quad t_\top = t\langle [\Sigma]^*, \dots, [\Sigma]^*, (0,\infty), \dots, (0,\infty)\rangle.$$

If $r \in \textsc{con}(t)$, then $L(t_\bot) \subseteq L(r) \subseteq L(t_\top)$.

Proof. We first show by induction on the size of templates that, for any template of the form $t\langle\circ\rangle$ and regular expressions r_1 and r_2, the relation $L(r_1) \subseteq L(r_2)$ implies that $L(t\langle r_1\rangle) \subseteq L(t\langle r_2\rangle)$. The base case is that if $t\langle\circ\rangle = \circ$, then

$$L(t\langle r_1\rangle) = L(r_1) \subseteq L(r_2) = L(t\langle r_2\rangle).$$

The inductive step consists of three cases:

- if $t\langle\circ\rangle = t_1\langle\circ\rangle \mid t_2$ for some templates t_1 and t_2, then

$$L(t\langle r_1\rangle) = L(t_1\langle r_1\rangle \mid t_2) = L(t_1\langle r_1\rangle) \cup L(t_2) \subseteq L(t_1\langle r_2\rangle) \cup L(t_2) = L(t\langle r_2\rangle);$$

- if $t\langle\circ\rangle = t_1\langle\circ\rangle t_2$ for some templates t_1 and t_2, then

$$L(t\langle r_1\rangle) = \{s_1s_2 | s_1 \in L(t_1\langle r_1\rangle) \land s_2 \in L(t_2)\} \subseteq \{s_1s_2 | s_1 \in L(t_1\langle r_2\rangle) \land s_2 \in L(t_2)\} = L(t\langle r_2\rangle);$$

- if $t\langle\circ\rangle = (t'\langle\circ\rangle)\{m, M\}$ for some m and M, then

$$\begin{aligned} L(t\langle r_1\rangle) &= \{s_1 \cdots s_c | m \le c \le M \land \forall i \le c.s_i \in L(t'\langle r_1\rangle)\} \\ &\subseteq \{s_1 \cdots s_c | m \le c \le M \land \forall i \le c.s_i \in L(t'\langle r_2\rangle)\} = L(t\langle r_2\rangle). \end{aligned}$$

Similarly, we can prove that, for any template of form $t\langle\triangleright, \triangleleft\rangle$, the relation $(m_1, M_1) \subseteq (m_2, M_2)$ implies that $t\langle(m_1, M_1)\rangle \subseteq t\langle(m_2, M_2)\rangle$. The trick here is that, assuming the quantified subexpression in $t\langle\triangleright, \triangleleft\rangle$ is r', we can substitute $r'\{\triangleright, \triangleleft\}$ with $\circ$ to get a new template $t'\langle\circ\rangle$ and then $L(t'\langle r'\{m_1, M_1\}\rangle) \subseteq L(t'\langle r'\{m_2, M_2\}\rangle)$ since $L(r'\{m_1, M_1\}) \subseteq L(r'\{m_2, M_2\})$.

For any regular expression $r = t\langle r_1, \dots, r_n, (m_1, M_1), \dots, (m_k, M_k)\rangle$, we denote $r^{(i)}$ the regular expression derived by substituting the first i holes (or quantifier holes if $i > n$) with $\emptyset$ (or $(1, 0)$ respectively). Note that $t_\bot = r^{(n+k)}$ and $r = r^{(0)}$. We prove that $L(r^{(i)}) \subseteq L(r^{(i-1)})$ for all i by induction on i. The base case can be shown by defining a new template $t'\langle\circ_i\rangle = t\langle\circ, r_2, \dots, r_n, (m_1, M_1), \dots, (m_k, M_k)\rangle$ with which we have $L(r^{(1)}) = L(t'\langle\emptyset\rangle) \subseteq L(t'\langle r_i\rangle) = L(r^{(0)})$ from the first claim we showed above. The inductive steps $L(r^{(i)}) \subseteq L(r^{(i-1)})$ can be shown in a similar way that defining a new template by substituting r_i with $\circ$ in $r^{(i-1)}$.

The other direction $L(r) \subseteq L(t_\top)$ can be proved in the same way. □

Intuitively, Theorem 4.3 states the following: if we treat the set $\textsc{con}(t)$ as a lattice with respect to language inclusion, then $t_\bot$ and $t_\top$ are the bottom and top elements of the lattice.[4] Using this property we obtain the following corollary, which allows us to efficiently discard templates that are guaranteed to not result in a repair.

Corollary 4.4. *Given a template t, a finite set of positive examples $P \subseteq \Sigma^*$, and a finite set of positive examples $N \subseteq \Sigma^*$, if $L(t_\bot) \cap N \neq \emptyset$ or $P \not\subseteq L(t_\top)$, there exists no regular expression $r \in \textsc{con}(t)$.*

Intuitively, to test whether a template t can result in a successful repair, we can just test whether $t_\bot$ rejects all negative examples and $t_\top$ accepts all positive examples.

[4]Notice that $r\{1, 0\}$ is equivalent to $\emptyset$, for every r.

4.2 Finding Simple Template Completions

In this section, we present the crucial idea that makes our repair procedure practical and propose two constraint-based encodings for finding whether there exists a solution for t in which holes of the form $\circ$ can only be replaced with sets of characters $[C]$.

Definition 4.5 (Simple Completion). Given a template t with holes $\circ_1, \ldots, \circ_n, (\triangleright_1, \triangleleft_1), \ldots, (\triangleright_k, \triangleleft_k)$, the set of *simple completions* of t is defined as $\textsc{conc}_C(t) = \{t\langle [C_1], \ldots, [C_n], (m_1, M_1), \ldots, (m_k, M_k))\rangle \mid C_i \subseteq \Sigma, m_j \in \mathbb{N}, M_j \in \mathbb{N} \cup \{\infty\}\}$.

Notice that, given a template t of cost k (w.r.t. to the initial regular expression), every simple completion that replaces each hole with a value different from that the one appearing in the original expression also has cost k.

First, finding a simple completion for a template is an NP-complete problem.

Theorem 4.6. *Given a template t a set of positive examples $P \in \Sigma^*$, and a set of negative examples $N \in \Sigma^*$, checking whether there exists a regular expression $r \in \textsc{conc}_C(t)$ consistent with P and N is an NP-complete problem.*

Proof. This problem is in NP since, given a regular expression, we can check if it is consistent with example sets P and N [Kilpeläinen and Tuhkanen 2003] and if it is in $\textsc{conc}_C(t)$.

We show that this problem is NP-hard by reducing 3-SAT to it. Recall that a 3-SAT problem is to determine the satisfiability of a formula in conjunctive normal form where each clause is limited to three literals. Consider a 3-SAT problem with formula

$$\phi = C_1 \wedge \cdots \wedge C_m,$$

where C_k is clause of form $(l_1^k \vee l_2^k \vee l_3^k)$ and a literal l_j^i is either x or $\neg x$ for some variable x. Let $X = \{x_i\}_{i=1}^n$ denote the variables in ϕ where n is the number of variables.

Now we construct a template t and two example sets N and P and then prove that ϕ is satisfiable iff there exists a completion of t consistent to N and P. Without losing of generality, we assume that $\Sigma = \{0, 1, 2\}$.

Before constructing t, we introduce an auxiliary template $t_{\mathbb{B}} := \Big((\circ_1\, \circ_2^*\, 2) \mid (\circ_3^*\, \circ_4\, 2)\Big)^*$. Intuitively, the template $t_{\mathbb{B}}$ represents Boolean variables. With a positive example 01020102 and a negative example 01121102, a completion of $t_{\mathbb{B}}$ consistent with these two examples can only be either in the set $\mathbb{T} := \{t_{\mathbb{B}}\langle\{0\}, \{0,1\}, C_3, C_4\rangle | 1 \notin C_3 \vee 0 \notin C_4\}$ or $\mathbb{F} := \{t_{\mathbb{B}}\langle C_1, C_2, \{0,1\}, \{0\}\rangle | 0 \notin C_1 \vee 1 \notin C_2\}$. If a completion $r \in \mathbb{F}$ (or $\mathbb{T}$), we say r assign false (or true, respectively) to the Boolean variable $t_{\mathbb{B}}$ represents.

We let the template $t := t_1 t_2$ to be the concatenation of two sub-template t_1 and t_2 where

$$t_1 \triangleq x_1 t_{\mathbb{B}} x_2 t_{\mathbb{B}} \cdots x_n t_{\mathbb{B}},$$

and

$$t_2 \triangleq l_1^1 t_{\mathbb{B}} l_2^1 t_{\mathbb{B}} l_3^1 t_{\mathbb{B}} \cdots l_1^m t_{\mathbb{B}} l_2^m t_{\mathbb{B}} l_3^m t_{\mathbb{B}}.$$

Intuitively, there is a $t_{\mathbb{B}}$ following each variable and literal and $t_{\mathbb{B}}$ represents the Boolean variables or literals it follows, e.g., the variable x_i should be set to true (or false) if the the completion of the $t_{\mathbb{B}}$ following x_i is in $\mathbb{T}$ (or $\mathbb{F}$, respectively).

Then we add five types of examples to P and N.

(1) A positive example $x_1\alpha \cdots x_n\alpha l_1^1\alpha \cdots l_3^m\alpha$ where $\alpha := 01020102$.
(2) A negative example $x_1\alpha \cdots x_{i-1}\alpha x_i\beta x_{i+1}\alpha \cdots l_3^m\alpha$ for each $i \in [1..n]$ where $\beta := 01121102$.
(3) A negative example $x_1\alpha \cdots x_n\alpha l_1^1\alpha \cdots l_j^i\beta \cdots l_3^m\alpha$ for each $i \in [1..m]$ and $j \in [1..3]$.

(4) For each literal $l^i_j = x_k$ with some variable x_k, two negative examples $x_1\alpha \cdots x_k\mathbf{0} \cdots l^1_1\alpha \cdots l^i_j\mathbf{1} \cdots l^m_3\alpha$ and $x_1\alpha \cdots x_k\mathbf{1} \cdots l^1_1\alpha \cdots l^i_j\mathbf{0} \cdots l^m_3\alpha$ where $\mathbf{0} := 01120112$ and $\mathbf{1} := 11021102$.
Similarly, for each literal $l^i_j = \neg x_k$ we add two negative examples $x_1\alpha \cdots x_k\mathbf{0} \cdots l^1_1\alpha \cdots l^i_j\mathbf{0} \cdots l^m_3\alpha$ and $x_1\alpha \cdots x_k\mathbf{1} \cdots l^1_1\alpha \cdots l^i_j\mathbf{1} \cdots l^m_3\alpha$;
(5) For each $i \in [1..m]$, we add a negative example $x_1\alpha \cdots l^i_1\mathbf{0}l^i_2\mathbf{0}l^i_3\mathbf{0} \cdots l^m_3\alpha$.

Then a complete expression $r \in \textsc{con}_C(t)$ is consistent with P and N iff r satisfies following constraints:

(1) For each $i \in [1..n]$, the completion of the template $t_{\mathbb{B}}$ following x_i is either in $\mathbb{T}$ or $\mathbb{F}$;
(2) For each $i \in [1..m]$ and $j \in [1..3]$, the completion of the template $t_{\mathbb{B}}$ following l^i_j is either in $\mathbb{T}$ or $\mathbb{F}$;
(3) For each literal $l^i_j = x_k$, the completion of the template $t_{\mathbb{B}}$ following x_k and the template $t_{\mathbb{B}}$ following l^i_j are in the same set $\mathbb{F}$ or $\mathbb{T}$. Similarly, for each literal $l^i_j = \neg x_k$, the completion of the template $t_{\mathbb{B}}$ following x_k and the template $t_{\mathbb{B}}$ following l^i_j are in different sets;
(4) For each $i \in [1..m]$ there exists a $j \in [1..3]$ such that the completion of the template $t_{\mathbb{B}}$ following l^i_j is in $\mathbb{T}$.

Note that the size of the template t and the size of the examples sets N and P are all polynomials of m. If ϕ can be satisfied by an assignment π, we can construct a regular expression consistent with N and P. For each variable $x_k \in X$, we complete the template $t_{\mathbb{B}}$ following x_k and the template $t_{\mathbb{B}}$ following any $l^i_j = x_k$ (or $\neg x_k$) with an expression in $\mathbb{T}$ (or $\mathbb{F}$) if $\pi(x_k) = 1$ (or 0, respectively). The resulting expression is in $\textsc{con}_C(t)$ and consistent with P and N.

On the other hand, if ϕ is unsatisfiable and there exists a regular expression $r \in \textsc{con}_C(t)$ consistent with P and N, we can extract an assignment π from r using the following construction such that π satisfies ϕ which leads to a contradiction: for each $k \in [1..n]$, let $\pi(x_k) = 1$ (or 0) if the completion of the template $t_{\mathbb{B}}$ following x_k is in $\mathbb{T}$ (or $\mathbb{F}$, respectively). The assignment π satisfies ϕ because of the property 4 of t, i.e., there is a $\mathbb{T}$ expression in each clause. □

We propose a pruning mechanism for simple template completions similar to the one presented in Section 4.1.

Theorem 4.7. *Given a template t with holes $\langle \circ_1, \ldots, \circ_n, (\triangleright_1, \triangleleft_1), \ldots, (\triangleright_k, \triangleleft_k)$, let t_Σ be defined $t\langle [\Sigma], \ldots, [\Sigma], (0, \infty), \ldots, (0, \infty)\rangle$. If $r \in \textsc{con}_C(t)$, then $L(r) \subseteq L(t_\Sigma)$.*

Proof. The same proof of Theorem 4.3 can be used to prove this theorem since $\{[C]\}$ is a subset of R and $[C] \subseteq [\Sigma]$ for any C. □

Theorem 4.7 states that, if the test t_Σ fails, no correct simple completion exists.

Example 4.8. Consider the example strings shown in Figure 1 and two templates $t^1 = \circ\{2\}[0-9]\{14\}$ and $t^2 = (\circ\circ)\{\triangleright, \triangleleft\}[0-9]\{14\}$ in Example 4.16. The expression $t^2_\Sigma = ([\Sigma][\Sigma])\{0, \infty\}[0-9]\{14\}$ accepts all positive examples in Figure 1. Also, there exists a simple completion $C_1 = [5]$ and $C_2 = [12345]$ such that $(C_1C_2)\{1\}[0-9]\{14\}$ is consistent with all examples. The expression $t^1_\Sigma = [\Sigma]\{2\}[0-9]\{14\}$ also accepts all positive examples in Figure 1. However there exists no simple completion for t^1 because, for any $[C] \subseteq \Sigma$, $C\{2\}[0-9]\{14\}$ must match both the positive example 5125632154125412 and the negative example 1525632154125412. □

If the t_Σ-test passes, we can move on to check whether a simple completion exists. Given the hardness of the problem (Theorem 4.6), we resort to an SMT-based approach and propose two SMT encodings with complementary complexities for the problem of finding simple completions. Our first SMT encoding (Section 4.2.1) generates constraints using the runs of the input examples on the automaton corresponding to the given regular expression. Our second encoding (Section 4.2.2) generates constraints using the inductive semantics of the regular expression.

4.2.1 Automaton-directed SMT Encoding. Our first technique is based on the following idea: Given a template t, we construct an automaton A_t that is parametric in the holes in t. For a string s, we can "run" s through the automaton A_t and collect information about what simple completions of the holes can cause s to be accepted by A_t. We use the collected information to generate a constraint Φ_s for which the satisfying assignment are simple completions that cause s to be accepted by the template t. In this section, we detail the construction of such an automaton and show how to generate constraints from it to find simple completions.

Automata. A *nondeterministic finite automaton with counters* K *and holes* $\circ_1, \ldots, \circ_n, (\triangleright_1, \triangleleft_1), \ldots, (\triangleright_k, \triangleleft_k)$ (NFAC) is a tuple $A = (Q, q_0, \delta, F)$ where Q is a finite set of states, $q_0 \in Q$ is an initial state, and $F \subseteq Q$ is a set of final states. Between any two states there is at most one transition and each transition $(q, l, q') \in \delta$ has one of the following labels l:

- $[C]$: a transition that can be crossed by any symbol in the set C;
- $\circ_i$: a transition that can be crossed by any symbol;
- ε: an ε-transition;
- k++: an ε-transition that increases the value of counter k by 1;
- $m \le k \le M/k \leftarrow 0$: an ε-transition that is triggered if the value of k is between m and M, and sets the value of k to 0;
- $\triangleright_i \le k \le \triangleleft_i/k \leftarrow 0$: an ε-transition that sets the value of k to 0;

Intuitively, the holes in the automaton are treated like their values in the test t_Σ. Given a string $s = a_0 \cdots a_n \in \Sigma^*$, an *$\epsilon$-loop-free accepting run* ρ of s in A is a tuple $(\bar{q}, f, g)$ where $\bar{q}$ is a sequence of states $q_0 \cdots q_N$, $f : [0..n] \mapsto [0..N]$ is a mapping from positions in the string to positions in the run, and $g : [0..N] \times K \mapsto \mathbb{N}$ is a mapping from positions in the run to counter values such that *i)* $q_N \in F$ is a final state, *ii)* for every $k \in K$, the initial counter value $g(0, k)$ is 0, *iii)* for every $i < i'$, we have $f(i) < f(i')$; *iv)* for every $i \le n$, there exists a set $C \subseteq \Sigma$ such that $(q_{f(i)}, [C], q_{f(i)+1}) \in \delta$ and $a_i \in C$; *v)* for every $j < N$ for which there does not exists an $i \le n$ such that $f(i) = j$, then $(q_j, l, q_{j+1}) \in \delta$ and l matches one of the following cases:

- $l = \varepsilon$: then for every $k \in K$, $g(j, k) = g(j + 1, k)$;
- $l = k$++: then $g(j, k) + 1 = g(j + 1, k)$ and for every $k' \ne k \in K$, $g(j, k') = g(j + 1, k')$;
- $l = m \le k \le M/k \leftarrow 0$: then $m \le g(j, k) \le M$, $g(j + 1, k) = 0$ and for every $k' \ne k \in K$, $g(j, k') = g(j + 1, k')$;
- $l = \triangleright \le k \le \triangleleft/k \leftarrow 0$: then $g(j + 1, k) = 0$ and for every $k' \ne k \in K$, $g(j, k') = g(j + 1, k')$;

Finally, for every $i < i'$ such that $q_i = q_{i'}$, there exists $i \le j < i$, such that $(q_j, \varepsilon, q_{j+1}) \notin \delta$—i.e., the run does not traverse ϵ-loops. Intuitively, the mapping $f(i)$ records a sequence of states traversed in the run when the i-th symbol is being read by the automaton. Given a string s, we use $\textsc{runs}_A(s)$ to denote the set of ϵ-loop-free accepting runs of s in A. We omit A when it is clear from context. Because we are considering only runs without ϵ-loops, the set of runs of a string is finite. In the following, we often just use accepting run instead of ϵ-loop-free accepting run. A string s is accepted by A if $\textsc{runs}_A(s) \ne \emptyset$.

Example 4.9. Figure 3 shows the NFAC corresponding to the template $\circ\{\triangleright, \triangleleft\}[0\text{-}9]\{14\}$. The NFAC has two counters k_1 and k_2. The string 5125632154125412 is accepted by A. The corresponding run is $r = q_0^0 q_1^1 q_2^2 q_0^3 q_1^4 q_2^5 q_0^6 q_3^7 q_4^8 q_5^9 \cdots q_4^{50} q_7^{51}$ (the superscript represents the position of the state in the run) with mapping $f(0) = 1$, $f(1) = 4$, $f(2) = 9$, $f(3) = 12$ and so on. The counter mapping

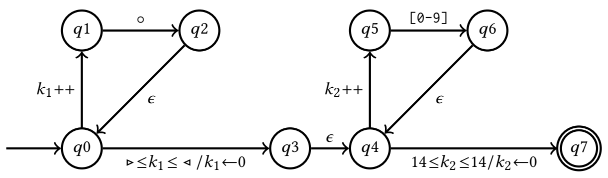


Fig. 3. Automaton constructed from the template ∘{▹, ◃}[0-9]{14}.

is $(g(0,k_1),\ldots,g(51,k_1)) = (0,1,1,1,2,2,2,0,\ldots,0)$, $g(i,k_2) = 0$ when $i < 9$ or $i = 51$, and $g(i,k_2) = \lceil (i-8)/3 \rceil$ when $9 \le i \le 50$. □

Template automata. We construct an NFAC A_t with set of counters K corresponding to the given template $t\langle \circ_1,\ldots,\circ_n,(\triangleright_1,\triangleleft_1),\ldots,(\triangleright_N,\triangleleft_N)\rangle$. For simplicity, we assume there exists one counter $k_{t'}$ for every sub-expression t' of t. The automaton A_t will accept all strings in t_Σ (as defined in Theorem 4.7). However, when the transitions labeled with holes are replaced with concrete values, the automaton accepts the language of the regular expression obtained by replacing the holes with those values. We illustrate the construction of A_t by induction on t. In the following, for every t_i we assume $A_{t_i} = (Q_i, q_0^i, \delta_i, F_i)$. The construction is similar to the Thompson construction from regular expressions to NFAs [Thompson 1968].

$A_{[C]} = (\{q_0,q_1\}, q_0, \{(q_0,C,q_1)\}, \{q_1\})$;
$A_\circ = (\{q_0,q_1\}, q_0, \{(q_0,\circ,q_1)\}, \{q_1\})$;
$A_{t_1|t_2} = (\{q_0\} \cup Q^1 \cup Q^2, q_0, \{(q_0,\varepsilon,q_0^1),(q_0,\varepsilon,q_0^2)\} \cup \delta^1 \cup \delta^2, F^1 \cup F^2)$;
$A_{t_1t_2} = (Q^1 \cup Q^2, q_0^1, \delta, F^2)$ where $\delta = \delta^1 \cup \delta^2 \cup \{(q,\varepsilon,q_0^2) \mid q \in F^1\}$;
$A_{t_1\{m,M\}} = (\{q_0,q_F\} \cup Q^1, q_0, \delta, \{q_F\})$ where

$$\begin{aligned}\delta &= \delta^1 \cup \{(q,\varepsilon,q_0) \mid q \in F^1\}\\ &\cup\ \{(q_0, k_{t_1}{+}{+}, q_0^1), (q_0, m \le k_{t_1} \le M/k_{t_1} \leftarrow 0, q_F)\};\end{aligned}$$

$A_{t_1\{\triangleright,\triangleleft\}} = (\{q_0,q_F\} \cup Q^1, q_0, \delta, \{q_F\})$ where

$$\begin{aligned}\delta &= \delta^1 \cup \{(q,\varepsilon,q_0) \mid q \in F^1\}\\ &\cup\ \{(q_0, k_{t_1}{+}{+}, q_0^1), (q_0, \triangleright \le k_{t_1} \le \triangleleft/k_{t_1} \leftarrow 0, q_F)\};\end{aligned}$$

The automaton in Figure 3 is the result of applying the construction to the template ∘{▹, ◃}[0-9]{14}. It is easy to see that the above construction places at most one transition between any two states.

As we observed, the language of the NFAC A_t is $L(t_\Sigma)$.

Lemma 4.10. *Given a template t, $L(A_t) = L(t_\Sigma)$.*

Proof. Transitions labeled with ∘ accept all characters in Σ which is the same as transitions labeled with [Σ]. Also, transitions labeled with $\triangleright \le k \le \triangleleft/k \leftarrow 0$ accept k with any value just like transitions labeled with $0 \le k \le \infty/k \leftarrow 0$ Then the proof is immediate since the construction of the automaton pretends that all holes are instantiated. with the substitution used in t_Σ. □

Since t_Σ accepts all positive examples, the above lemma guarantees that there exists at least one run for each positive example.

Constraint generation. We now formally describe how, given a string $s = a_0 \cdots a_n \in \Sigma^*$ and an accepting run $\rho = (q_0,\ldots,q_N,f,g)$ of s in A_t, we generate an SMT constraint encoding the values of all simple completions that make the run accepting.

Our constraint will have the following variables. For every hole $\circ \in H_\circ$ and for every symbol $a \in \Sigma$, we will have a Boolean variable $x_\circ^a$—i.e., given a solution to the generated constraints, the hole $\circ$ can be replaced with a character class $[C]$ containing all the characters for which $x^a_\circ$ is set to true. For every quantifier hole $(\triangleright, \triangleleft) \in H_{\triangleright\triangleleft}$, we will have positive integer variables $x_\triangleright$ and $x_\triangleleft$ denoting the values of the holes—i.e., the holes $\triangleright$ and $\triangleleft$ can be replaced by any values satisfying the constraint we generate.

When generating constraints for templates of the form $t\{\triangleright, \triangleleft\}$, the possible values of the hole $\triangleright$ will depend on whether the quantified expression t can accept the empty string ε. In particular, if t accepts ε, any run that traverses the template k consecutive times can be turned into a run that traverses the same template $k + c$ times by taking c ϵ-loops. When this is the case, the value of $\triangleright$ is essentially unconstrained. Given a template t we define the Boolean variable $\textsc{eps}_t$ that is true if t accepts ε. The value of $\textsc{eps}_t$ may depend on the values of quantifiers and the following constraints capture this aspect ($\top$ and $\bot$ stand for true and false, respectively).

$$\textsc{eps}_\emptyset = \bot \quad \textsc{eps}_\varepsilon = \top \quad \textsc{eps}_{[C]} = \bot \quad \textsc{eps}_\circ = \bot \quad \textsc{eps}_{t_1 t_2} = \textsc{eps}_{t_1} \wedge \textsc{eps}_{t_2}$$
$$\textsc{eps}_{t_1 | t_2} = \textsc{eps}_{t_1} \vee \textsc{eps}_{t_2} \quad \textsc{eps}_{t_1\{m,M\}} = \textsc{eps}_{t_1} \vee m = 0 \quad \textsc{eps}_{t_1\{\triangleright,\triangleleft\}} = \textsc{eps}_{t_1} \vee x_\triangleright = 0.$$

Given a sub-expression $t'\{\triangleright, \triangleleft\}$, we define $\text{v}_{\triangleright,\triangleleft} \subseteq \mathbb{N}$ as the set of values that counter $k_{t'}$ has when the run exits the automaton corresponding to $t'\{\triangleright, \triangleleft\}$:

$$\text{v}_{\triangleright,\triangleleft} = \{g(i, k_{t'}), \mid (q_i, \triangleright \leq k_{t'} \leq \triangleleft / k_{t'} \leftarrow 0, q_{i+1}) \in \delta\}$$

The following constraints describe what hole values cause the automaton to accept the run:

- for a hole $\circ \in H_\circ$:
$$\varphi_\circ \equiv \bigwedge \{x^{a_j}_\circ \mid \exists i.(q_i, q_{i+1}) \in m_\circ(\circ) \wedge f(j) = i\};$$
- for a sub-expression $t'\{\triangleright, \triangleleft\}$ with hole $(\triangleright, \triangleleft) \in H_{\triangleright\triangleleft}$:
$$\varphi_{\triangleright,\triangleleft} \quad \equiv \quad [\textsc{eps}_{t'} \vee \bigwedge_{k \in \text{v}_{\triangleright,\triangleleft}} x_\triangleright \leq k] \wedge [\bigwedge_{k \in \text{v}_{\triangleright,\triangleleft}} k \leq x_\triangleleft];$$

A constraint for a run ρ has to deal with all the holes:

$$\Phi_\rho \equiv \bigwedge_{\circ \in H_\circ} \varphi_\circ \wedge \bigwedge_{(\triangleright,\triangleleft) \in H_{\triangleright\triangleleft}} \varphi_{\triangleright,\triangleleft}$$

Example 4.11. We illustrate our constraint generations technique using the string `5125632154125412` and the NFAC A from Figure 3. Note that there is one hole $\circ_1$ and one pair $(\triangleright_1, \triangleleft_1)$ of quantifier holes in A. This input only has one run ρ in A and the corresponding hole constraints are $\Phi_{\circ_1} = x^5_{\circ_1} \wedge x^1_{\circ_1}$, and $\Phi_{\triangleright_1,\triangleleft_1} = [\bot \vee x_{\triangleright_1} \leq 2] \wedge [2 \leq x_{\triangleleft_1}]$. Finally, $\Phi_\rho = \Phi_{\circ_1} \wedge \Phi_{\triangleright_1,\triangleleft_1} = x^5_{\circ_1} \wedge x^1_{\circ_1} \wedge (x_{\triangleright_1} \leq 2) \wedge (2 \leq x_{\triangleleft_1})$. □

A string is accepted if it has at least one successful run:

$$\Phi_s \equiv \bigvee_{\rho \in \textsc{runs}_A(s)} \Phi_\rho$$

Finally, the following constraint guarantees that the solution is consistent with all the examples:

$$\Phi^A_{P,N} \equiv \bigwedge_{s \in P} \Phi_s \wedge \bigwedge_{s \in N} \neg\Phi_s$$

A solution to the generated constraints is a valuation function τ assigning values to all the variables to make the constraint true. Given a solution τ, we define the corresponding regular expression as:

$$t_\tau = t\langle [C_1], \ldots, [C_n], (x_{\triangleright_1}, x_{\triangleleft_1}), \ldots, (x_{\triangleright_k}, x_{\triangleleft_k}) \rangle$$

where $C_i = \{a \mid \tau(x^a_{\circ_1}) = \top\}$.

Note that our SMT encoding does not complete a quantifier hole with a Kleene star (*) because, in the constraints, the values of holes of the form $\triangleleft$ are only allowed to be positive integer numbers. However, whenever the lower bound of a quantifier hole is completed with a 0, and the higher bound is a number greater than the length of the longest example, the completion for the hole is equivalent to a * with respect to the given examples. For example, given positive examples b, bbbb and a negative example cc, the template b{$\triangleright$, $\triangleleft$} could be instantiated as b{0, 5}. Since the longest example is of length 4, the completion b{0, 5} is equivalent to b* with respect to matching the given examples.

We are now ready to state our correctness theorem.

Theorem 4.12 (Soundness and completeness). *Given a template t, a set of positive examples $P \in \Sigma^*$, and a set of negative examples $N \in \Sigma^*$, there exists a regular expression $r \in \textsc{conc}_C(t)$ iff $\Phi^A_{P,N}$ is satisfiable. Moreover, if τ is a solution to $\Phi^A_{P,N}$, then $t_\tau \in \textsc{conc}_C(t)$.*

Proof. We first show that, if $\Phi^A_{P,N}$ is satisfiable, then for every satisfying assignment τ the expression t_τ is consistent with P and N. For each positive example $s \in P$, there exists a path $\rho \in \textsc{runs}_A(s)$ such that Φ_ρ is satisfied by τ. Since $L(A_{t_\tau}) = L(t_\tau)$, the accepting run ρ is the witness that s is accepted by t_τ. Similarly, for each negative example $s \in N$, every run of A_{t_τ} on s is not an accepting run since all Φ_ρ is false under τ.

In the other direction, if there exist a regular expression $r = t\langle[C_1], \dots, [C_n], (m_1, M_1), \dots, (m_k, M_k)\rangle \in \textsc{conc}_C(t)$ consistent with P and N, we can construct an assignment τ_r satisfying $\Phi^A_{P,N}$ from r. For each variable $x^a_{\circ_i}$, we set $\tau_r(x^a_{\circ_i}) = \top$ iff $a \in C_i$. For each pair of integer variables $x_{\triangleright_i}$ and $x_{\triangleleft_i}$, we set $\tau_r(x_{\triangleright_i}) = m_i$ and $\tau_r(x_{\triangleleft_i}) = M_i$. For each positive example $s \in P$, there is an accepting run ρ of A_r on s which implies that Φ_ρ is satisfied by τ_r. Similarly, for each negative example $s \in N$, there exists no accepting run of A_r on s, which implies that $\textsc{runs}_{A_r}(s) = \emptyset \Rightarrow \Phi_s = \bot$. □

Complexity. The automaton-directed SMT encoding generates a constraint that has size linear in the number of runs of the examples, but does not depend on the size of the regular expression. If the input template has h holes and each of the e input examples has at most c accepting runs, the constraint has size $O(hec)$. In general, the number of runs of an input example can be exponential in his length—i.e., if l is the length of the longest example the constraint has size $he2^{O(l)}$. In the above analysis we ignored the size of the alphabet, if the alphabet has size y, the constraints contain $O(hy)$ variables. While in practice y can be very large, the constraints only mention a variable $x^a_\circ$ if *i)* the character a appears in at least one input example, *ii)* the character a traverses the hole $\circ$ in the automaton for some run. Hence, the number of variables does not blow up in practice.

Example 4.13. Consider a template $t = (\circ_1\circ_2^*)^*$. For the string $s = 12345$ there are 16 runs of A_t. In general, there are 2^{m-1} runs of A_t on strings with length m. □

4.2.2 *Regular-expression-directed SMT Encoding.* The automaton-directed SMT encoding can become impractical for expressions for which strings can have many accepting runs. In this section, we present a different SMT encoding that avoids this shortcoming by directly using the semantics of regular expressions to decide whether a string is accepted.

Our constraints will use the same set of variables from the constraints presented in Sec. 4.2.1 as well as some new variables. Given a template t, a valuation τ assigning values to all the holes, a string $s = a_0 \dots a_n \in \Sigma^*$, and positions $i, j \leq n$, we define a Boolean variable $t(s[i,j])$ such that

$$t(s[i,j]) = \top \iff a_i \dots a_j \in L(t_\tau)$$

In other words, $t(s[i,j])$ denotes whether t_τ—i.e., the expression obtained by replacing the holes with the corresponding variable values—accepts the substring of s that starts at index i and ends at index j. By this definition, the string s is accepted by t if and only if $t(s[0,n]) = \top$. The constraints for $t(s[i,j])$ with $0 \le i, j \le n$, are defined inductively below. For every quantified sub-expressions $t\{m, M\}$ and $t\{\triangleright, \triangleleft\}$, the formalization also includes variables $t\{c\}(s[i,j])$ for every value $0 \le c \le n$—i.e., for t repeated c times.

- $\emptyset(s[i,j]) = \bot$ and $\varepsilon(s[i,j]) = \bot$.
- $[C](s[i,j]) = \top$ iff $i = j$ and $a_i \in C$.
- $\circ(s[i,j]) = x_\circ^{a_i} \wedge i = j$—i.e., the string a_i is in the language of the hole $\circ$ iff the character a_i is included in the solution for the hole.
- if $t = t_1|t_2$ then $t(s[i,j]) = t_1(s[i,j]) \vee t_2(s[i,j])$.
- if $t = t_1 t_2$ then

$$t(s[i,j]) = [\bigvee_{i \le k < j} t_1(s[i,k]) \wedge t_2(s[k+1,j])] \vee [\textsc{eps}_{t_1} \wedge t_2(s[i,j])] \vee [t_1(s[i,j]) \wedge \textsc{eps}_{t_2}]$$

- if $t = t'\{m, M\}$ then
  - $(t'\{1\})(s[i,j]) = t'(s[i,j])$
  - for every $2 \le c \le min(n, M)$:

$$(t'\{c\})(s[i,j]) = \bigvee_{i \le k < j} (t'\{c-1\})(s[i,k]) \wedge t'(s[k+1,j])$$

  - if $\textsc{eps}_{t'}$: $t(s[i,j]) = \bigvee_{1 \le c \le min(n,M)} (t'\{c\})(s[i,j])$
  - if $\neg\textsc{eps}_{t'}$:

$$t(s[i,j]) = \bigvee_{min(n,m) \le c \le min(n,M)} (t'\{c\})(s[i,j])$$

- if $t = t'\{\triangleright, \triangleleft\}$
  - $(t'\{1\})(s[i,j]) = t'(s[i,j])$
  - for every $2 \le c \le n$:

$$(t'\{c\})(s[i,j]) = \bigvee_{i \le k < j} (t'\{c-1\})(s[i,k]) \wedge t'(s[k+1,j])$$

  - if $\textsc{eps}_{t'}$: $t(s[i,j]) = \bigvee_{1 \le c \le j-i+1} c \le x_\triangleleft \wedge (t'\{c\})(s[i,j])$
  - if $\neg\textsc{eps}_{t'}$:

$$t(s[i,j]) = \bigvee_{1 \le c \le j-i+1} x_\triangleright \le c \le x_\triangleleft \wedge (t'\{c\})(s[i,j])$$

Example 4.14. We take the string s = `5125632154125412` and the template $t = \circ\{\triangleright, \triangleleft\}$`[0-9]{14}` to illustrate the encoding. First of all, the template t is the concatenation of two templates $t_1 = \circ\{\triangleright, \triangleleft\}$ and t_2 = `[0-9]{14}`. We illustrate some of the constraints. For the sub-expression `[0-9]`, for every i, we have `[0-9]`$(s[i,i]) = \top$ since all characters in s are digits. Next, we show an example of how a constraint is added for a quantifier:

$$\texttt{[0-9]}\{2\}(s[1,2]) = \texttt{[0-9]}\{1\}(s[1,1]) \wedge \texttt{[0-9]}(s[2,2])$$

As expected, this encoding produces a constraint that is equi-satisfiable to the one produced by the automaton-directed encoding. □

The following constraint guarantees that the solution is consistent with all examples:

$$\Phi^r_{P,N} \equiv \bigwedge_{s \in P} t(s[0, |s|-1]) \wedge \bigwedge_{s \in N} \neg t(s[0, |s|-1])$$

We are now ready to state our correctness theorem.

Theorem 4.15 (Soundness and completeness). *Given a template t, a set of positive examples $P \in \Sigma^*$, and a set of negative examples $N \in \Sigma^*$, there exists a regular expression $r \in \textsc{conc}_C(t)$ if and only if $\Phi^r_{P,N}$ is satisfiable. Moreover, if τ is a solution to $\Phi^r_{P,N}$, then $t_\tau \in \textsc{conc}_C(t)$.*

Proof. It suffice to show that

$$t(s[i,j]) = \top \iff a_i \dots a_j \in L(t_\tau),$$

with which we have $\Phi^r_{P,N} \iff \bigwedge_{s\in P} s \in L(t_\tau) \wedge \bigwedge_{s\in N} s \notin L(t_\tau)$.

For the base cases that $t = \emptyset$, $[C]$ or $\circ$, it is quite straight that $t(s[i,j]) = \top \iff a_i \dots a_j \in L(t_\tau)$. Then inductively,

- if $t = t_1 \mid t_2$, then
$$t(s[i,j]) = t_1(s[i,j]) \vee t_2(s[i,j]) = \top \iff s[i,j] \in L(t_{1,\tau}) \vee s[i,j] \in L(t_{2,\tau});$$
- if $t = t_1t_2$, then t_τ accepts $s[i,j]$ iff there exists two substrings s_1 and s_2 such that $s[i,j] = s_1s_2$, $t_{1,\tau}$ accepts s_1 and $t_{2,\tau}$ accepts s_2. Note that the choice of k in the definition of $t_1t_2(s[i,j])$ is choosing the position to split $s[i,j]$ to two substrings, and we can choose one substring to be ϵ if one of the sub-expression can accept ϵ;
- if $t = t'\{m, M\}$, then t_τ accepts $s[i,j]$ iff we can split $s[i,j]$ to $c \in [m..M]$ number of substrings such that each substring can be accepted by t'_τ, which is consistent to the definition of $t'\{m, M\}(s[i,j])$;
- if $t = t'\{\triangleright, \triangleleft\}$, then t_τ accepts $s[i,j]$ iff we can split $s[i,j]$ to $c \in [\tau(x_\triangleright)..\tau(x_\triangleleft)]$ number of substrings such that each substring can be accepted by t'_τ, which is consistent to the definition of $t'\{\triangleright, \triangleleft\}(s[i,j])$.

□

Complexity. The regular-expression-directed SMT encoding generates a constraint that has size linear in the size of the template and cubic in the length of the input examples. The cubic factor comes from the constraints generated for quantified templates. In particular, since in the variables $(t\{c\})(s[i,j])$, the elements c, i and j can assume values between 1 and n, there will be n^3 variables. If the input template has size k and each of the e input examples has length at most l, the generated constraint has size $O(kel^3)$. Unlike what we saw for the constraints generated by our automaton-based encoding, this complexity bound is also a lower bound. Therefore, the regular-expression-directed encoding generates larger constraints than the automaton-directed encoding when the number of runs r of the automaton on the input strings is smaller than l^3.

4.3 Template generation

When our algorithm cannot find a simple completion for a template t, it uses three operations to generate more templates from it: *i)* add a new hole by either replacing a character class $[C]$ with $\circ$, or by replacing a quantifier $\{m, M\}$ with $\{\triangleright, \triangleleft\}$. *ii)* expand an existing hole $\circ$ by replacing it with $\circ_1\circ_2$, $\circ_1|\circ_2$, or $\circ_1\{\triangleright_1, \triangleleft_1\}$. *iii)* reduce an existing hole $\circ$ by replacing its parent node with a hole.

Example 4.16. In this example, we run RFixer on the example $r = [C_1]\{2\}[C_2]\{14\}$ (with $C_1 =$ `[51|52|53|54|55]` and $C_2 =$ `[0-9]`) showed in Figure 1.

The initial set of templates is the following:

1. $\circ\{2\}[C_2]\{14\}$ *2.* $[C_1]\{\triangleright, \triangleleft\}[C_2]\{14\}$ *3.* $[C_1]\{2\} \circ \{14\}$ *4.* $[C_1]\{2\}[C_2]\{\triangleright, \triangleleft\}$

RFixer extracts the template $t_1 = \circ\{2\}[C_2]\{14\}$, tries to compute a simple completion for it, and fails. Then, RFixer expands the hole in the template t_1 to enumerate:

5. $\circ \circ \{2\}[C_2]\{14\}$ *6.* $(\circ|\circ)\{2\}[C_2]\{14\}$ *7.* $\circ\{\triangleright, \triangleleft\}\{2\}[C_2]\{14\}$

and adds a new hole in t_1 to enumerate:
8. $\circ\{2\}\circ\{14\}$ *9.* $\circ\{\triangleright,\triangleleft\}[C_2]\{14\}$ *10.* $\circ\{2\}[C_2]\{\triangleright,\triangleleft\}$
Eventually RFixer reaches $t = (\circ_1\circ_2)\{\triangleright,\triangleleft\}[C_2]\{14\}$ in the search and tries to compute a simple completion for it. RFixer successfully finds a simple completion by replacing $\circ_1$ with [5], $\circ_2$ with [12345] and $\{\triangleright,\triangleleft\}$ with {1 , 1}. □

4.4 Correctness

We can now state the correctness theorem for RFixer.

Theorem 4.17. *Given a regular expression r, a finite set of positive examples $P \subseteq \Sigma^*$, and a finite set of negative examples $N \subseteq \Sigma^*$ such that $P \cap N = \emptyset$, Algorithm 1 is sound and complete for the repair-from-examples problem (see Definition 3.3)—i.e., RFixer always outputs a regular expression at minimal distance from r that is consistent with the examples.*

Proof. First, notice that if r' is a solution to the repair problem of cost k, there exists a template t of cost k from which r' can be obtained as a simple completion. Concretely, let τ_r be the AST for r and let $\tau_r[\tau_1'/\tau_1, \dots, \tau_n'/\tau_n]$ be the edit of minimum cost k that transforms τ_r into τ_r'. Then, replacing every leaf in each subtree τ_i' with a hole generates a template of cost k for which r' is a possible simple completion.

Second, we show that the algorithm explores all templates in increasing cost. The algorithm initially explores all templates obtained by inserting one hole in the leaves of the original expression. These are exactly all the templates at distance 2 from the original expression r (notice that no template can have distance less than 2 from the original expression according to our distance definition). Next, we can observe that the template generation described in this section is such that *i)* all templates are eventually enumerated (we cache intermediate results and never explore the same template twice), *ii)* the generated templates have costs higher than the template they are being generated from. The second property, together with the fact that the data structure Q in Algorithm 1 extracts templates in order of cost ensures that a template of cost k is visited only if all templates of cost smaller than k have been already explored, which guarantees minimality. □

5 OPTIMIZATIONS AND IMPROVEMENTS

In this section, we present a set of optimizations that further improve the performance of RFixer.

5.1 Avoiding Equivalent Templates

Our implementation employs a set of optimizations that limit the exploration of syntactically and semantically equivalent templates. First, our algorithm uses simple forms of hashing to avoid repeating syntactically equivalent templates that might result from expanding the same set of holes in different orders. To limit the set of semantically equivalent templates, we exploit the associativity of the concatenation and union operators and the commutativity of union. Whenever a hole $\circ$ is expanded as $\circ_1\circ_2$ (resp. $\circ_1|\circ_2$) we will not expand $\circ_2$ into a template of the form $\circ_3\circ_4$ (resp. $\circ_3|\circ_4$). Because every template of the form $t_1(t_2t_3)$ (resp. $t_1|(t_2|t_3)$) is equivalent to the expression $(t_1t_2)t_3$ (resp. $(t_1|t_2)|t_3$), our pruning strategy preserves completeness. For commutativity, whenever a hole $\circ$ is expanded (in multiple steps) into an expression of the form $t_1|t_2$, we only keep and expand templates where the size of t_1 is greater or equal than the size of t_2. Notice that if we discard this template, we will still explore the template $t_2|t_1$, which is semantically equivalent and satisfies the size requirement. Hence, our pruning strategy still preserves completeness.

5.2 Avoiding Redundant Tests

The next optimization uses properties of regular expressions to avoid repeating some of the template pruning tests from Section 3 and simple-completion queries when a hole is expanded. For example, consider a template t such that $t_\top$ accepts all the positive examples. If we expand some hole $\circ$ using an alternation $\circ_1|\circ_2$, the resulting template t' will be such that $t_\top$ still accepts all the positive examples. Moreover, if the constraint $\Phi_{P,N}$ generated for t for finding simple completions was not satisfiable, the constraint for t' will also be unsatisfiable. Table 4 summarizes when a test can be skipped. Notice that the tests $t_\top$ and $t_\bot$ only need to be performed when creating the initial set of templates. Formally, a $t_\top$ test can be skipped every time because the languages of the expressions $\Sigma^*\Sigma^*$, $\Sigma^*|\Sigma^*$, and $\Sigma^*\{0,\infty\}$ are all equivalent to Σ^*. Similarly, the $t_\bot$ test can be skipped because $\emptyset\emptyset$, $\emptyset|\emptyset$, and $\emptyset\{1,0\}$ are all equivalent to $\emptyset$. The t_Σ test can be skipped for $\circ_1|\circ_2$ but not $\circ_1\{\triangleright,\triangleleft\}$, because $\Sigma|\Sigma$ is equivalent to Σ but $\Sigma\{0,\infty\}$ is a superset of Σ.

Expansion	$t_\top$	$t_\bot$	t_Σ	$\Phi_{P,N}$
$\circ_1\circ_2$	skip	skip		
$\circ_1\|\circ_2$	skip	skip	skip	skip
$\circ_1\{\triangleright,\triangleleft\}$	skip	skip		

Fig. 4. Tests to skip upon a hole expansion.

5.3 Generalizing Solutions using Quantitative Objectives

Our notion of distance from Definition 3.1 only captures the number of AST edits to a regular expression and does not differentiate with respect to different template completions. In particular, two simple completions of the same template will have the same distance from the original expression. Consider the following example. Assume that for the template $\circ\{\triangleright,\triangleleft\}$ and some set of examples P and N the following two regular expressions are both possible solutions for the template: r_1 = [0-9]{0,∞} = [0-9]$*$ and r_2 = [0236]{2,8}. The regular expression r_1 is simpler, more general, and easier to understand, but our notion of distance will not prefer it and will consider the two solutions of equal cost. In this section, we present an approach for preferring "more general" repairs. In particular, we show how we can use MaxSMT optimization objectives—i.e., SMT formulas with a minimization objective over a set of weights—to produce better character sets and simpler lower and upper bounds for quantifiers.

In regular expressions, character classes are used to represent common sets of characters. Common character classes are [a-z], [A-Z], [0-9], \s, and \w. In our implementation, we add constraints to encode such character classes. For example, for a certain hole $\circ$, we can encode the possibility of including the character class [0-9] in its concretization by adding a variable $x_\circ^{\texttt{[0-9]}}$ together with the constraint $x_\circ^{\texttt{[0-9]}} \rightarrow x_\circ^{\texttt{0}} \wedge \ldots \wedge x_\circ^{\texttt{9}}$. The constraint guarantees that, if the class is added to the template, all its characters are also added. Next, we show how to add weights to the constraints so that a completion using a character class is preferred to one that uses multiple individual characters. First, an example. For the character class [0-9], we assign the following weight to a variable $w_\circ^{\texttt{[0-9]}}$: $x_\circ^{\texttt{[0-9]}} \rightarrow w_\circ^{\texttt{[0-9]}} = -17$. For each individual digit character $d \in \texttt{[0-9]}$, we assign $x_\circ^d \rightarrow w_\circ^d = 2$. The minimization objective is $w_\circ^{\texttt{[0-9]}} + \sum_{d\in[0,9]} w_\circ^d$. With this particular constraint, the whole character class [0-9] (cost $20 - 17 = 3$) will be more expensive than a single digit (cost 2) but cheaper than two digits (cost 4). In the same way, we can add constraints to character classes [a-z] and [A-Z] such that a single character class is preferred to multiple lowercase/uppercase letters respectively. Finally, we add a constraint such that \w is preferred to two or more of [0-9], [a-z], and [A-Z].

Next, we add constraints that prefer commonly used quantifiers—i.e., ?, $*$, and $+$. We do so by introducing a weight variable $w_\triangleright$ (resp. $w_\triangleright$) for each hole of the form $\triangleright$ (resp. $\triangleleft$). The variable $w_\triangleright$ is assigned a value 1 whenever $\triangleright$ is assigned a value greater than 1. The value of $w_\triangleright$ is 0 otherwise. The variable $w_\triangleleft$ is assigned a value 1 whenever $\triangleleft$ is assigned a value different than 1 or ∞—i.e., a value larger than the length of the longest example. The value of $w_{\triangleright}$ is 0 otherwise. We then ask the algorithm to find the solution that minimizes the sum of all weights across all constraints.

Note that we only run a MaxSMT query if we have already found an SMT solution to a template. Because an individual MaxSMT query has negligible running time, this optimization does not affect the overall runtime of RFixer.

6 EVALUATION

We implemented RFixer in Java using Z3 version 4.6.2 as the SMT solver for the generated constraints. All experiments were performed on a machine with 2.50GHz Intel(R) Xeon(R) Platinum 8175M CPU with 6.90GB MaxHeapSize for JVM and we used a timeout value of 300s. We evaluated the effectiveness of our technique using a series of experiments to answer the following research questions:

Q1 Can RFixer repair regular expressions from examples and how does it compare to state-of-the-art tools? (§ 6.2)
Q2 Can RFixer produce high-quality repairs? (§ 6.3)
Q3 Which of the two SMT encodings (Sec. 4.2.1 and 4.2.2) has better performance? (§ 6.4)
Q4 How much search space can RFixer prune? (§ 6.5)

6.1 Benchmarks

Our benchmarks have three sources: *i)* the Automata Tutor dataset [Tutor 2015], *ii)* the RegExLib library [RegExLib 2017], and *iii)* the regular expression repair work by Rebele et al. [2018].

Automata Tutor. Automata Tutor is a website for helping students learn automata theory and regular expressions [D'Antoni et al. 2015b]. The website allows teachers of such topics to create homework assignments and assign them to students to solve. The students can then submit solutions to such assignments, and the tool will automatically check whether the solution is correct. For assignments related to automata constructions, Automata Tutor can also automatically fix the student solutions and use this fix to provide useful feedback to the student [D'Antoni et al. 2015a]. This capability is not there for regular expressions, as no way to automatically repair regular expressions have been proposed prior to the current paper. Many regular-expression assignments (29 individual assignments in total) and student-submitted solutions from Automata Tutor are publicly available in an anonymized dataset [Tutor 2015]. Our second set of benchmarks is extracted from this dataset. For each of the unique 2,104 incorrect student submissions present in the dataset, we used the correct regular expression to create a set of test cases on which the student submission is failing and used it as a specification for the repair problem. Concretely, we used conformance testing techniques for finite state machines [Yannakakis 1991] to generate enough inputs to traverse each state in the automata corresponding to the correct and incorrect regular expressions. This technique generates a number of examples that is cubic in the numbers of states of the minimal DFA corresponding to the regular expression and linear in the size of the alphabet. The regular expressions in this benchmark set have length varying between 1 and 100 (avg. 14) and operate over small alphabets (2 symbols). The number of examples per expression varies between 4 and 96. We don't evaluate the MaxSMT optimization presented in Section 5 on these benchmarks because they do not allow character classes and numerical quantifiers.

RegExLib. RegExLib [RegExLib 2017] is a website collecting regular expressions from hundreds of developers around the world. The website currently contains thousands of regular expressions and most of them come equipped with an English description of what the expression is supposed to do and a few test cases. Each expression then receives a rating from 1 to 5 from the website

Table 1. Number of solved benchmarks by solver

Technique	RegExLib	Automata Tutor	[Rebele et al. 2018]
RFixer-a	19	1,336	30
RFixer-r	21	1,568	29
RFixer	23	1,588	35
RegexGenerator++	3	—	—
[Rebele et al. 2018]	—	—	50
Total	25	2,104	50

users. In general, it is not possible for us to evaluate our tool on all of the expressions on the website because there is no specification of how expressions should be repaired. However, we found several instances in which the expressions were deemed to be imprecise by some user and where the user provided some counterexamples. We then collected 25 expressions using the following methodology: *i)* we examined the expressions in chronological order (from most recently submitted), *ii)* we skipped expressions that used non-regular operators—e.g., positive lookahead or boundary symbols—and overly complicated expressions—i.e., with length greater than 200, *iii)* we skipped expressions with fewer than five comments, low ratings, and expressions that did not have comments containing counterexamples, *iv)* due to the fair amount of manual inspection required to browse the expressions and their comments, we stopped at 25 expressions. The regular expressions in this benchmark set have character-length varying between 3 and 192 (avg. 59) and operate over the ASCII alphabet set (128 characters). The number of examples per expression varies between 10 and 39. The expressions cover several categories of strings: numbers (6), time/date (5), email/urls (6), and other tasks (8).

Rebele et al. [2018]. Rebele et al. recently proposed a tool for "adding positive examples" to regular expressions. Given a regular expression r and a set of positive examples P not accepted by the expression r, their tool uses heuristics to greedily modify the input regular expressions r and generate a r' that matches the examples in P. Due to the different problem formulation, we do not compare against this tool on the other benchmarks discussed earlier. We collected the 50[5] regular expressions presented by Rebele et al. and compare RFixer against their tool. One interesting aspect of this benchmark set is that each expression comes with a *training set*—i.e., a set P of positive and negative examples we are using to repair the original expression—and a *test set*—i.e., a set of positive and negative examples the repaired expression will be evaluated on. A good repair should perform well on the test set. The regular expressions in this benchmark set have length varying between 15 and 725 (avg. 90.0) and operate over the ASCII alphabet set (128 characters). The number of examples per expression varies between 17 and 75. The expressions cover several categories of strings: numbers (24), time/date (16), email/urls (5), and other tasks (5).

6.2 Effectiveness of RFixer

In this experiment, we use our benchmarks to evaluate how effective RFixer is at producing repaired expressions that are consistent with the given examples. For each benchmark, we run both the automaton- and regular-expression-directed encodings and report whether at least one of the two encodings solved the benchmark. When both encodings succeed, we report the minimum of their running times as if the encodings were executed in parallel.

[5] The original benchmark set contains 52 regular expressions, but we do not consider two of them because they use the end-of-string anchor character \$, which our implementation currently does not support.

Automata Tutor. RFixer successfully repaired 1,588/2,104 (75%) regular expressions (with respect to the given examples) using one of the two SMT encodings (avg. time: 9.3s). We are not aware of repair tools that can handle restricted alphabets and could not compare against other tools.

RegExLib. For these benchmarks, we compared RFixer against RegexGenerator++ [Bartoli et al. 2016], a state-of-the-art tool that uses genetic programming to *synthesize* regular expressions based on examples (we are not aware of repair tools that can handle both positive and negative examples). RegexGenerator++ starts from an initial seed regular expression and then evolves it into a larger population of regular expressions using crossover and mutation operations. After a number of generations, the tool outputs the regular expression with the highest fitness. We adapted RegexGenerator++ into a *repair* tool by using the regular expression being repaired as the initial seed (using the default seed resulted in worse results). RegexGenerator++ was executed with the following (default) parameter values: threads number = 4, population size = 500, maximum number of generations = 1000, percentage of number generations = 20.0. While RFixer repaired 23/25 (92%) expressions using one of the two SMT encodings (avg. time: 7.3s), RegexGenerator++ only produced 3/25 (12%) repairs that were correct on all the examples (avg. time: 198s). Moreover, the expressions produced by RegexGenerator++ are completely different from the initial ones. For example, for the regular expression from Section 2, RFixer produced the repaired expression `5[12345][0-9]{14}`, while RegexGenerator++ produced the expression `\w[123]\14|\w427654851354895|[555][555]\w\w++`.

Rebele et al. [2018]. For these benchmarks, we compared RFixer against the state-of-the-art tool from Rebele et al. [2018]. We use Reb to refer to this tool. RFixer repaired 35/50 (70%) expressions using one of the two SMT encodings (avg. time: 2.4s). Since Reb does not impose minimality constraints on the size of the repair[6], it successfully repaired 50/50 (100%, avg. time: 0.2s). Since Reb uses a heuristic greedy search, it does not provide minimality guarantees. Because of the complexity of regular expression operators—e.g., commutativity of union—we could not automatically measure the distance of the Reb repairs from the original expressions. However, we manually inspected the repairs produced by Reb and, in general, they most of the times appeared to have a much higher distance from the original expression than those produced by RFixer. For example, on the input expression `[0-3][0-9]( *)[a-z|A-Z|  ]+[0-9]{4}` RFixer produced the repair `[1A-Z]\w(  *)[\-\w,  ]+[\-\d|]{4}` while Reb produced the repair `([0-3][0-9])?(    *|19-|97-|86-|91-)?(([a-z|A-Z| ](21,|27,|6,)?)+)?[0-9]{2}(-|\||-8-)?[0-9]\|?[0-9](-7)?`. We will see in Section 6.3 how minimality with respect to our distance definition can result in repairs that better generalize to unseen examples.

Summary. To answer **Q1**, **RFixer can effectively produce minimal regular expression repairs (using examples) across different applications** that existing tools cannot handle or for which they provide no minimality guarantees.

RFixer could not solve benchmarks that *i)* required very large fixes—e.g., changing the expression in many places, or *ii)* involved complex nested quantifiers, which caused both SMT encodings as well as the $t_\top$ test to be practically slow.

6.3 Quality of Computed Repairs

In this section, we show that repairing regular expressions from examples can also lead to high-quality repairs with respect to left-out test sets or more complex specifications. For each of our

[6]When we discard the minimality constraint, the repair from example problem becomes trivial as one can simply produce any regular expression consistent with the examples.

benchmark sets, we design experiments intended at assessing whether RFixer can find repairs that are in some sense "good".

Automata Tutor. We use the fact that for these benchmarks we have access to the regular expressions given by the instructor to turn RFixer into a tool for finding expressions equivalent to the correct solutions. For each benchmark, we run a counterexample-guided inductive synthesis (CEGIS) algorithm as follows. First, we use RFixer to find an expression r' that is consistent with an initial set of examples E (the same examples used to in Section 6.2). Second, we check whether r' is semantically equivalent to the expression r given by the instructor. If it is, we successfully terminate, otherwise, we generate a counterexample on which the two expressions differ, add it to E, and call RFixer again.

Using the CEGIS algorithm, RFixer successfully found *correct* repairs for 1,156/2,104 (55%) expressions using one of the two SMT encodings (avg. time: 9.4s). The reduction in the number of solved benchmarks with respect to Section 6.2 is due to the increased search depth required to find expressions consistent with the additional examples. Of the solved benchmarks, 847 returned a correct expression using just the initial set of examples and the remaining 309 required on average 1.7 extra examples to find a correct repair. The 432 expressions for which RFixer found a repair from examples but not a correct repair had added on average 1.4 extra examples when they timed out. The expressions for which additional examples caused a timeout were usually cases where "fixing" the original expressions on the given initial examples would require a small repair, but adding the new examples explored by the CEGIS algorithm caused the minimal repair to be very far from the original expression. For these cases, the initial examples were usually not representative enough and they either did not provide enough useful information to guide the search of completion for a template, or they provided redundant information which led to significant overhead for the constraint solver.

RegExLib. For these benchmarks, we do not have access to a specification and could not formally assess the correctness of the produced repair. To assess whether the produced repairs were "good", two of the paper's authors independently manually inspected the repairs produced by RFixer together with the given examples and assigned one of the following labels to each of them: *i) correct* the repair appears to correctly match the intention conveyed by the examples; *ii) correct template* the repair is incorrect, but it is using a template that can lead to a correct completion (i.e., we need more examples to discover the correct completion); *iii) incorrect* the repair is using a template that cannot result in a good completion. Label disagreements (approximately 15% of the labels) were resolved through discussions between the authors. In this evaluation, we also considered the Max-SMT optimization described in Section 5.3 to see if this optimization could result in "better" repaired expressions. Hence, we have four variants of RFixer: two SMT encodings with or without the Max-SMT optimization. For 13/23 of the benchmarks RFixer could solve, at least one of the four variants of RFixer produced a repair labeled as *correct.*[7] For 3 of the remaining 11 cases, at least one of the four variants of RFixer produced a repair labeled as *correct template.* For the remaining 7 cases, RFixer produced repairs labeled as *incorrect.*

We observed that RFixer could not find a *correct* repair when one of the following scenarios happened. The first scenario is when the original expression is too far from a correct solution. For example, in `YAGO-date2`, the original expression is `[0-3]0-9[a-z|A-Z| ]+[0-9]{4}`, which captures dates in the form of `19 July 1808`. However, `1986-08-29` also appears as a positive example and this example has a dramatically different structure from the ones captured by the

[7] In two cases, the two SMT encodings produced different results (one *correct* and one *correct template*) due to the existence of multiple solutions. We report these two cases as *correct.*

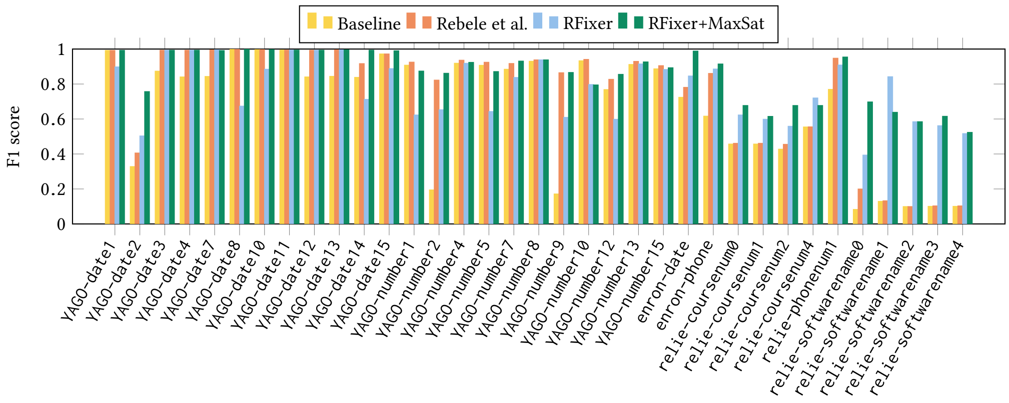


Fig. 5. F1 scores for the benchmarks from Rebele et al. [2018] for which RFixer terminated.

expression. This difference in structure causes RFixer to find minor variations of the original expression that overfit to the given example. The second scenario in which RFixer cannot find correct repairs is one in which examples are not representative enough. For example, in `relie-coursenum2`, the original expression is `[A-Za-z]{4,10} \d{3}`, where `[A-Za-z]{4,10}` intends to match the course name and `\d{3}` intends to match the course number. Existing positive and negative examples have very similar structures (e.g., `Math 175` is positive and `Pages 394` is negative), and for these examples RFixer does not generalize well to unseen positive examples (e.g., `Rackham 575`).

In summary, RFixer produced high-quality repairs for 57% of the solved benchmarks and identified the right repair template for 13% of the benchmarks. For all of the repairs labeled as *incorrect*, a correct repair would have required a very complex change to the regular expression—e.g., in 6/7 of these cases the correct repair required synthesizing complex expressions to restrict the ranges of certain multi-digit numbers, such as changing the month field of a date from `\d{2}` to `0?[1-9]|1[12]`. The MaxSMT optimization had very limited effect on the quality of the RegExLib benchmarks.[8] For one expression, RFixer *with* the MaxSMT optimization produced a *correct* repair by generalizing to a character class whereas RFixer *without* the optimization produced only a *correct template*. For two expressions, RFixer *without* the MaxSMT optimization produced a *correct* repair whereas RFixer *with* the optimization produced only a *correct template* because it overgeneralized a character class or a quantifier.

Rebele et al. [2018]. Each expression in this benchmark set comes with a *training set* and a *test set*—i.e., a set of examples not used at repair time. A repair obtained using the training set is "good" it if performs well well on the test set. Following the approach presented by Rebele et al. [2018], we use the F1 score—i.e., a number between 0 and 1 computed as the average of precision and recall—to evaluate the accuracy of the repairs on the test set (higher values of F1 are better). The results are shown in Figure 5.

We start by considering the version of RFixer in which the MaxSMT optimization described in Section 5.3 is enabled because these are likely to generalize to unseen inputs. For cases in which both the automaton- and the regular-expression-directed encodings terminated, we report the averages of their F1 scores. RFixer produced repairs that, on average, improved the F1 scores with respect to the test set by 0.16 (113% average improvement). Only for 4/35 expressions the F1 scores decreased. Across the problems solved by both RFixer and Reb, RFixer produced repairs that, on

[8] In 1 case, when using the MaxSMT optimization, one of the automaton-directed encoding produced a *correct* repair while the regular-expression-encoding produced a *correct template* due to the existence of multiple solutions. We report this case as *correct*.

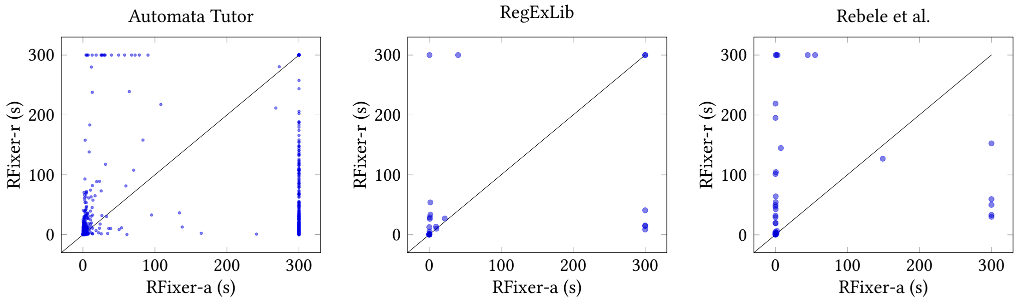


Fig. 6. Performance: RFixer-a vs RFixer-r

average, had F1 scores that were 0.07 higher than those of the repairs produced by Reb (56% average improvement). Only for 8/35 expressions the F1 scores of RFixer's repair were lower than those of Reb.

When considering the version of RFixer in which the MaxSMT optimization was disabled, the resulting F1 scores were on average 10.9% lower than those obtained using the MaxSMT optimization. This result indicates that the MaxSMT optimization is beneficial for producing repairs that generalize beyond the given examples.

Although RFixer can solve fewer benchmarks than Reb, RFixer clearly produces higher-quality repairs that those produced by Reb thanks to its minimality guarantees. In particular, RFixer's F1 scores are 1.56X better than those of the state-of-the-art tool Reb!

Summary. This experiment showed that for regular expressions over small alphabets, RFixer can use a variant of the CEGIS algorithm to effectively synthesize expressions that are semantically equivalent to target regular expressions. Moreover, for real world regular expressions, RFixer can produce high-quality repairs that generalize beyond the examples given as specification. Finally, the MaxSMT optimization technique described in Section 5.3 allows RFixer to produce repairs that better generalize to unseen data.

To answer **Q2**, **RFixer can *i)* produce correct repairs for regular expressions with small alphabets** and with a reference specification, and ***ii)* produce higher-quality repairs than those produced by existing tools** for real-world ASCII regular expressions.

6.4 Effectiveness of Different SMT Encodings

In this experiment, we compare the performance of the automaton-directed (RFixer-a) and regular-expression-directed (RFixer-r) SMT encodings presented in Sections 4.2.1 and 4.2.2, respectively. Figure 6 compares the running times of RFixer-a and RFixer-r on our benchmarks using scatter plots (points above the diagonal indicate that RFixer-a is faster than RFixer-r).

For the Automata Tutor benchmarks on which both solvers terminate, RFixer-r is, on average, 2.0X slower than RFixer-a. However, RFixer-a times out on 252 benchmarks that RFixer-r can solve and RFixer-r times out on 20 benchmarks that RFixer-a can solve. For the RegExLib benchmarks on which both solvers terminate, RFixer-r is, on average, 4.5X slower than RFixer-a. RFixer-a times out on 4 benchmarks that RFixer-r can solve and RFixer-r times out on 2 benchmarks that RFixer-a can solve. For the benchmarks from Rebele et al. [2018] on which both solvers terminate, RFixer-r is, on average, 1.75X slower than RFixer-a. RFixer-a times out on 5 benchmarks that RFixer-r can solve and RFixer-r times out on 5 benchmarks that RFixer-a can solve.

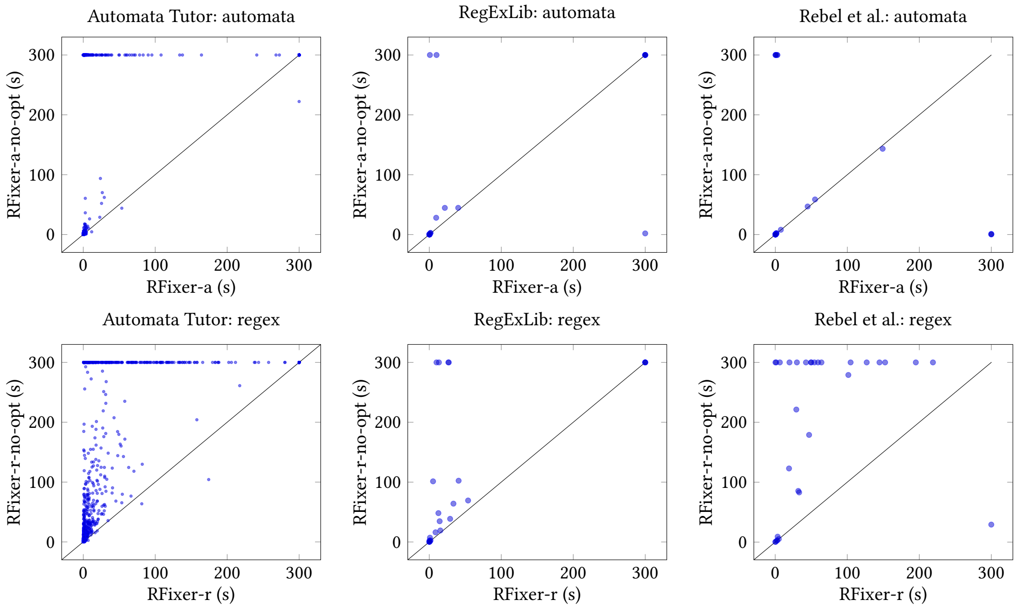


Fig. 7. Effectiveness of RFixer with and without optimizations

In general, RFixer-a times out when the expressions result in templates for which the examples have exponentially many runs and RFixer-r times out for benchmarks that cause cubic complexity incurred by the SMT encoding to become impractical. A deployment of RFixer for a practical application can run these two encodings in parallel and return the first solution returned by either encoding.

To answer **Q3**, **the two SMT encodings are both beneficial and can solve different sets of benchmarks**, but in general RFixer-r is slower than RFixer-a.

6.5 Effectiveness of Optimizations

In this experiment, we evaluate the effectiveness of the template pruning and search reduction techniques described in Sections 4.1, 5.1, and 5.2. We use RFixer-a-no-opt and RFixer-r-no-opt to denote the variants of RFixer-a and RFixer-r in which all these optimizations are turned off—i.e., these variants enumerate every possible template in increasing order and run an SMT query on every template. The results are shown in Figure 7 (points above the diagonal indicate that the optimized version is faster).

For both RFixer-a and RFixer-r, the optimized versions of the search are consistently faster (avg. 49% and 79% respectively) than the non-optimized ones when both the optimized and non-optimized versions terminate. RFixer-a can solve 336 Automata Tutor problems, 2 RegExLib problems, and 5 problems from Rebele et al. [2018] that RFixer-a-no-opt cannot solve. Interestingly, RFixer-a-no-opt can solve one RegExLib problem that RFixer-a cannot solve. This particular benchmark causes one of the $t_\top$ tests to trigger an exponential behavior for the regular expression matching algorithm used by Java. RFixer-r can solve 263 Automata Tutor problems, 4 RegExLib problems, and 17 problems from Rebele et al. [2018] that RFixer-r-no-opt cannot solve.

For Automata Tutor problems that could be solved with and without optimizations, RFixer explores an average of 18.5 templates whereas RFixer-no-opt explores 104.9—i.e., on average, RFixer prunes 82.4% of the templates. For the RegExLib problems, RFixer explores an average of 61.7 templates before finding a solution whereas RFixer-no-opt explores an average of 293.0 templates—i.e., on average, RFixer prunes 78.9% of the templates. For the problems from Rebele et al. [2018] RFixer explores an average of 1.9 templates before finding a solution whereas RFixer-no-opt explores an average of 152.3 templates—i.e., on average, RFixer prunes 98.8% of the templates. For the problems that RFixer-no-opt could not solve, RFixer explored an average of 200.4 templates.

To answer **Q4**, **RFixer's optimizations are effective and crucial for performance**.

7 RELATED WORK

Pruning Spaces in Program Synthesis. One of the key challenges in program synthesis is to efficiently prune the large search space of possible programs [Gulwani et al. 2017] and many techniques have been proposed to do so. Enumerative [Udupa et al. 2013] and version-space-algebra synthesis techniques [Gulwani 2011; Gulwani et al. 2012; Singh and Gulwani 2016] enumerate programs and avoid enumerating syntactically and semantically equivalent terms. RFixer focuses on regular expressions while existing tools focus on imperative or functional programs. Hence, RFixer can use properties of regular expressions to prune the search space by: *i)* using template pruning to eliminate templates, *ii)* using templates to represent infinitely many regular expressions and relying on SMT solvers to efficiently search the space of solutions for a template.

Learning Regular Expressions. The problem of automatically generating regular expressions from examples has been explored in many domains [Alquezar and Sanfeliu 1994; Bartoli et al. 2014, 2016; Dupont 1996; Fernau 2005; Galassi and Giordana 2005; Lee et al. 2016]. Existing approaches are based on domain-specific heuristics, tackle restricted forms of regular expressions, or do not provide any completeness guarantees. We discuss the ones that are closest to our work.

AlphaRegex [Lee et al. 2016] is an enumeration algorithm for synthesizing simple regular expressions over small alphabets from examples. AlphaRegex does not support complex quantifiers or large alphabets (all the synthesized expressions are over alphabets of size 2). AlphaRegex enumerates templates of increasing size and uses a template pruning technique similar to ours for pruning regular expressions that will not result in correct solutions. Their approach does not involve quantifier holes and uses simpler variants of the two tests $t_\perp$ and $t_\top$. Our approach extends the template pruning idea from AlphaRegex in multiple directions. First, we introduce quantified holes and devise ways to perform template pruning for such holes. Second, we present a new test t_Σ that is unique to our symbolic search based on simple completions. Third, we propose a new range of optimizations in Section 5 that avoid performing template pruning tests when expanding holes. Aside from the differences in template pruning, the biggest contribution of RFixer when compared to AlphaRegex is the support for real world regular expression with quantifiers and character classes enabled by our SMT based approach. AlphaRegex directly enumerates characters and does not use symbolic search, which is why it is limited to small alphabets. Even if AlphaRegex was able to somehow effectively enumerate characters, synthesizing the character class `[a-zA-Z]` in AlphaRegex would require the enumeration to discover an expression `a|b|...|z|A|...Z`, which is already 5 times larger than the largest expression AlphaRegex can synthesize.

RegexGenerator++ [Bartoli et al. 2014, 2016] is a state-of-the-art engine for synthesis of regular expressions from examples. RegexGenerator++ first extracts some seed regular expression from the examples and then uses genetic programming to mutate the initial expressions and search for further expressions. First, RegexGenerator++ is not guaranteed to produce a correct solution—i.e., one that is correct on all examples. Second, RegexGenerator++ may produce solutions of arbitrary size. Our approach is very different from RegexGenerator++ in that it solves a repair problem instead of a synthesis one and it provides minimality, soundness, and completeness guarantees.

Repairing Regular Expressions. ReLIE [Li et al. 2008] proposes a technique for repairing regular expressions that accept "too many" strings, while Rebele et al. [2018] propose a technique for adding "missing words" to a regular expression. Given a regular expression and a new set of *negative* examples, the goal of ReLIE is to modify the original expression so that it rejects the new examples. Similarly, given a regular expression and a new set of *positive* examples, the goal of Rebele et al. is to modify the original expression so that it accepts the new examples. Neither of these tools can handle both positive and negative examples. The two aforementioned tools use a set of heuristics to search for a modified regular expression that accepts/rejects the new examples by allowing a limited set of transformations—e.g., add/remove a disjunct from a union, or augment/reduce the set in a character class. Unlike RFixer, these tools do not provide any minimality guarantees and may produce regular expressions that are very different from the original ones. As we showed in Section 6.3, due to its minimiality guarantees, RFixer produces repairs that generalize to left-out data much better than the repairs produced by Rebele et al. [2018] (ReLIE is not publicly available and we cannot compare against it).

CrowdBoost [Cochran et al. 2015] is a genetic programming approach for repairing regular expressions using crowd-sourcing. CrowdBoost uses genetic programming operators over the DFA representation of the regular expression and requires a crowd (i.e., paid online workers) to classify examples that different expressions in the set of candidates disagree on. The final output is a DFA. While the DFA can be converted back to a regular expression, the construction would yield regular expressions that are completely different from the original expression.

To our knowledge, RFixer is the first sound and complete tool that can repair practical regular expressions from examples while providing guarantees on the size of the generated repair.

8 CONCLUSION

We presented RFixer, the first tool that can repair regular expressions with character classes and numerical quantifiers using examples. Given a regular expression and sets of positive and negative examples, RFixer finds the closest regular expression to the initial one that correctly classifies the examples. RFixer's repair algorithm uses enumerative search with new symbolic techniques for effectively pruning the search space. The main new feature of RFixer is a symbolic treatment of the problem of searching for numerical quantifiers and characters in the large alphabet space employed by regular expressions. For this problem, RFixer relies on SMT solvers. Our evaluation shows that RFixer can repair regular expressions from a variety of domains and that RFixer produces high-quality repairs that generalize to examples outside the ones given as part of the specification. Our preliminary results show that RFixer could be used in a variety of applications. First, RFixer can be deployed in tools like Automata Tutor [D'Antoni et al. 2015a] and used to build feedback systems for helping students understand regular expressions. Second, RFixer can be deployed as a helping tool for debugging regular expressions in systems like https://regexr.com/, which provide intuitive interfaces for people to experiment with regular expressions.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their feeback. This work was supported, in part, by NSF under grants CNS-1763871, CCF-1750965, CCF-1744614, and CCF-1704117; and by the UW-Madison OVRGE with funding from WARF.

REFERENCES

2018. COMPSCI 194 - LEC 016, https://bcourses.berkeley.edu/courses/1267848/pages/regex. https://bcourses.berkeley.edu/courses/1267848/pages/regex

R. Alquezar and A. Sanfeliu. 1994. Incremental Grammatical Inference From Positive And Negative Data Using Unbiased Finite State Automata. In *In Proceedings of the ACL'02 Workshop on Unsupervised Lexical Acquisition*. 291–300.

Alberto Bartoli, Giorgio Davanzo, Andrea De Lorenzo, Eric Medvet, and Enrico Sorio. 2014. Automatic Synthesis of Regular Expressions from Examples. *Computer* 47, 12 (Dec. 2014), 72–80. https://doi.org/10.1109/MC.2014.344

Alberto Bartoli, Andrea De Lorenzo, Eric Medvet, and Fabiano Tarlao. 2016. Inference of Regular Expressions for Text Extraction from Examples. *IEEE Trans. Knowl. Data Eng.* 28, 5 (2016), 1217–1230. https://doi.org/10.1109/TKDE.2016.2515587

Philip Bille. 2005. A survey on tree edit distance and related problems. *Theoretical Computer Science* 337, 1 (2005), 217 – 239. https://doi.org/10.1016/j.tcs.2004.12.030

Robert A. Cochran, Loris D'Antoni, Benjamin Livshits, David Molnar, and Margus Veanes. 2015. Program Boosting: Program Synthesis via Crowd-Sourcing. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*. 677–688. https://doi.org/10.1145/2676726.2676973

Loris D'Antoni, Dileep Kini, Rajeev Alur, Sumit Gulwani, Mahesh Viswanathan, and Björn Hartmann. 2015a. How Can Automatic Feedback Help Students Construct Automata? *ACM Trans. Comput.-Hum. Interact.* 22, 2 (2015), 9:1–9:24. https://doi.org/10.1145/2723163

Loris D'Antoni, Matthew Weavery, Alexander Weinert, and Rajeev Alur. 2015b. Automata Tutor and what we learned from building an online teaching tool. *Bulletin of the EATCS* 117 (2015). http://eatcs.org/beatcs/index.php/beatcs/article/view/365

Pierre Dupont. 1996. Incremental regular inference. In *Grammatical Interference: Learning Syntax from Sentences*, Laurent Miclet and Colin de la Higuera (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 222–237.

Henning Fernau. 2005. Algorithms for Learning Regular Expressions. In *Proceedings of the 16th International Conference on Algorithmic Learning Theory (ALT'05)*. Springer-Verlag, Berlin, Heidelberg, 297–311. https://doi.org/10.1007/11564089_24

Ugo Galassi and Attilio Giordana. 2005. Learning Regular Expressions from Noisy Sequences. In *Abstraction, Reformulation and Approximation*, Jean-Daniel Zucker and Lorenza Saitta (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 92–106.

E Mark Gold. 1978. Complexity of automaton identification from given data. *Information and Control* 37, 3 (1978), 302 – 320. https://doi.org/10.1016/S0019-9958(78)90562-4

Sumit Gulwani. 2011. Automating string processing in spreadsheets using input-output examples. In *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011*. 317–330. https://doi.org/10.1145/1926385.1926423

Sumit Gulwani, William R. Harris, and Rishabh Singh. 2012. Spreadsheet data manipulation using examples. *Commun. ACM* 55, 8 (2012), 97–105. https://doi.org/10.1145/2240236.2240260

Sumit Gulwani, Oleksandr Polozov, and Rishabh Singh. 2017. Program Synthesis. *Foundations and Trends in Programming Languages* 4, 1-2 (2017), 1–119. https://doi.org/10.1561/2500000010

A.J.G. Hey, S. Tansley, and K.M. Tolle. 2009. *The Fourth Paradigm: Data-intensive Scientific Discovery*. Microsoft Research. https://books.google.com/books?id=oGs_AQAAIAAJ

Pekka Kilpeläinen and Rauno Tuhkanen. 2003. Regular Expressions with Numerical Occurrence Indicators - preliminary results. In *Proceedings of the Eighth Symposium on Programming Languages and Software Tools, SPLST'03, Kuopio, Finland, June 17-18, 2003*, Pekka Kilpeläinen and Niina Päivinen (Eds.). University of Kuopio, Department of Computer Science, 163–173.

Mina Lee, Sunbeom So, and Hakjoo Oh. 2016. Synthesizing regular expressions from examples for introductory automata assignments. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences, GPCE 2016, Amsterdam, The Netherlands, October 31 - November 1, 2016*, Bernd Fischer and Ina Schaefer (Eds.). ACM, 70–80. https://doi.org/10.1145/2993236.2993244

Yunyao Li, Rajasekar Krishnamurthy, Sriram Raghavan, Shivakumar Vaithyanathan, and H. V. Jagadish. 2008. Regular Expression Learning for Information Extraction. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing (EMNLP '08)*. Association for Computational Linguistics, Stroudsburg, PA, USA, 21–30. http://dl.acm.org/citation.cfm?id=1613715.1613719

Thomas Rebele, Katerina Tzompanaki, and Fabian M. Suchanek. 2018. Adding Missing Words to Regular Expressions. In *Advances in Knowledge Discovery and Data Mining*, Dinh Phung, Vincent S. Tseng, Geoffrey I. Webb, Bao Ho, Mohadeseh Ganji, and Lida Rashidi (Eds.). Springer International Publishing, Cham, 67–79.

RegExLib. 2017. Regular Expression Library. http://regexlib.com/.

Rishabh Singh and Sumit Gulwani. 2016. Transforming spreadsheet data types using examples. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA,* *January 20 - 22, 2016*. 343–356. https://doi.org/10.1145/2837614.2837668

Ken Thompson. 1968. Programming Techniques: Regular Expression Search Algorithm. *Commun. ACM* 11, 6 (June 1968), 419–422. https://doi.org/10.1145/363347.363387

Automata Tutor. 2015. Data from the tool Automata Tutor. https://github.com/AutomataTutor/automatatutor-data.

Abhishek Udupa, Arun Raghavan, Jyotirmoy V. Deshmukh, Sela Mador-Haim, Milo M.K. Martin, and Rajeev Alur. 2013. TRANSIT: Specifying Protocols with Concolic Snippets. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '13)*. 287–296.

Mihalis Yannakakis. 1991. Testing Finite State Machines. In *Proceedings of the Twenty-third Annual ACM Symposium on Theory of Computing (STOC '91)*, David Lee (Ed.). ACM, New York, NY, USA, 476–485. https://doi.org/10.1145/103418.103468