Repairing Regex-Dependent String Functions

Nariyoshi Chida
nariyoshichidamm@gmail.com
NTT Social Informatics Laboratories
Japan

Tachio Terauchi
terauchi@waseda.jp
Waseda University
Japan

ABSTRACT

Regex-dependent string functions are string functions that take regular expressions (regexes) as parameters and are popular means of manipulating strings. They are frequently used for, e.g., string transformation and substring search. Despite the importance, writing these functions is far from easy. To rectify this situation, recent research made significant progress by proposing automated methods for synthesizing regexes based on Programming by Examples (PBE). However, there still is a gap between these methods and the goal of synthesizing regex-dependent string functions. First, the existing methods focus on whole-string matching, whereas most regex-dependent string functions adopt substring matching. Second, the existing methods focus only on the regex, but many commonly used regex-dependent string functions, such as replace and replaceAll, also take as parameter a *replacement* to specify how the substrings matched to the regex will be replaced.

This paper fills the gap by presenting the *first* PBE-based method for repairing regex-dependent string functions. Like the recent methods for regex synthesis, our algorithm builds on enumerative search with pruning and SMT constraint solving, but with extensions to support substring matching and replacement. The main challenge is the large search space. We address the challenge by novel ideas such as incorporation of *origin information* in examples to identify the locations of substrings to be matched, a new *substring-context-aware* pruning technique, and a novel use of SMT constraints to insert captures that can be referred from the replacement. Additionally, we identify a novel necessary and sufficient condition that can be used to detect and filter unrepairable instances. We implemented our algorithm as a prototype tool called R2-DS and evaluated it on real-world benchmarks. Results show that our algorithm efficiently repairs the bugs in the real world and finds high-quality repairs.

CCS CONCEPTS

• **Software and its engineering** → **Software notations and tools**; • **Theory of computation** → **Regular languages**.

KEYWORDS

Regular Expression, Programming by Example, Program Repair

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).
ASE '24, October 27-November 1, 2024, Sacramento, CA, USA
© 2024 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-1248-7/24/10
https://doi.org/10.1145/3691620.3695005

ACM Reference Format:
Nariyoshi Chida and Tachio Terauchi. 2024. Repairing Regex-Dependent String Functions. In *39th IEEE/ACM International Conference on Automated Software Engineering (ASE '24), October 27-November 1, 2024, Sacramento, CA, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3691620.3695005

1 INTRODUCTION

Modern programming languages provide various built-in string functions that take regular expressions (regexes) as parameters. Such *regex-dependent string functions* include repalceAll, which replaces all substrings matched to the given regex with possibly-different strings, and search, which finds a substring matched to the given regex. These regex-dependent string functions play an important role in string manipulations in programs. For example, they are used for sanitizing untrusted substrings [33, 34], parsing [11], and extracting substrings [29, 31]. Despite the importance, writing regex-dependent string functions is far from an easy task and is a common source of bugs in software [23, 32, 43].

In response, researchers have developed automated methods for synthesizing regexes based on Programming By Examples (PBE) [12, 17, 18, 25, 28, 35, 42, 45]. However, despite the advances made by these works, there is still a gap between them and synthesizing regex-dependent string functions.

Substring Matching: Although the prior works focus on whole string matching, most regex-dependent string functions adopt substring matching. Therefore, even if we synthesize a regex using the existing methods, there is no guarantee that the regex matches the intended substrings when we use it through regex-dependent string functions.

Replacement: Commonly used regex-dependent string functions for string transformations such as replace and replaceAll take as parameter a *replacement* to specify how to replace the substrings matched by the given regex. Since the behavior of replaceAll depends on both the regex and the replacement, it is not possible to synthesize regex-dependent string functions by focusing only on one of them.

In this paper, we fill the gap by proposing the *first* PBE-based algorithm for repairing regex-dependent string functions. For space, most of the paper focuses on replaceAll, one of the most popular regex-dependent string functions [11]. However, the core ideas are easily applicable to the other regex-dependent string functions such as replace and search (cf. Sec. 6). Our algorithm takes a set of *examples* that describe the desired input-output of the function to be synthesized, and the goal is to find a pair of a regex and a replacement such that replaceAll using the pair as the parameters behaves consistently with the examples. To bias the solution to the user-intended one, our algorithm can be given a "pre-repair" pair of a regex and a replacement as an additional input so that the goal

is not only to find the parameters consistent with the examples but to find one that is syntactically close to the given pre-repair ones.

Like the recent methods for regex synthesis, our algorithm builds on enumerative search with pruning and SMT constraint solving, but with extensions to support substring matching and replacement. The main challenge is the large search space. We address the challenge by novel ideas such as incorporation of *origin information* in examples to identify the locations of substrings to be matched, a new *substring-context-aware* pruning technique, and a novel use of SMT constraints to insert captures that can be referred from the replacement. Additionally, we identify a novel necessary and sufficient condition that can be used to detect and filter unrepairable instances.

We have implemented our algorithm as a tool called R2-DS (Repairer for Regex-Dependent String functions). R2-DS is designed for JavaScript. This is needed because different regex engines often have different semantics that give different behavior to regex-dependent string functions, and we have to pick one for concreteness. However, we think that the core ideas of the paper are applicable to regex-dependent string functions in others such as Python and Java. We evaluated R2-DS on real-world benchmarks collected from GitHub and StackOverflow. Results show that R2-DS can find a high-quality repair efficiently. In summary, we make the following contributions.

- We introduce the replaceAll-repair problem, which is the first problem of repairing replaceAll function from examples. Concretely, we incorporate *origin information* in examples to help specifying the user's intention and reasoning about the repair easier (Sec. 4.1), define a novel *origin semantics* of replaceAll functions (Sec. 4.2), and with them, formally define the repair problem (Sec. 4.3).
- We show that, in contrast to the repair problems for regexes studied in prior works, our repair problem has instances that do not have a solution. Additionally, we show a novel necessary and sufficient condition for our repair problem instance to have a solution (Sec. 4.4).
- We propose the first PBE-based algorithm for repairing replaceAll functions (Sec. 5). Our algorithm supports substring matching and replacement. Especially, our algorithm incorporates a new substring-context-aware pruning technique and a novel use of SMT constraints to insert captures. We show that our ideas are applicable to the other regex-dependent string functions such as replace and search (Sec. 6).
- We implement our algorithm in a tool called R2-DS and evaluate it on real-world benchmarks (Sec. 7). Results show that R2-DS can efficiently repair the real-world bugs of regex-dependent string functions. We also conduct an ablation study which shows usefulness of the newly introduced techniques of context-aware pruning and SMT-constraint-based insertion of captures.

2 OVERVIEW

In this section, we give an example, inspired by StackOverflow posts [39, 40], to illustrate how the users represent their intentions using our examples, and how R2-DS solves the repair problem.

Motivating Example. The user wants to replace all substrings of the form $[x]y[x]$, where x and y are strings, with $x : y$, unless preceded by the character #. For example, the user wants to replace the text "[a]ab[a], [b]c[a]" with "a:ab, [b]c[a]" and the text "#[b]de[b], [a]c[a]" with "#[b]de[b], a:c". For this purpose, the user prepared the regex $(?{<}!\#)\backslash[(_1.^*)_1\backslash].^*\backslash[\backslash1\backslash]$ and the replacement \$1:\$2 as parameters of replaceAll. Roughly, replaceAll functions behave like the following. For an input string w with the regex *reg* and the replacement *rep*, it tries to match *reg* at each position of w from left to right. If the matching succeeds, it replaces the matched substring with a string obtained from *rep*, and otherwise, it does nothing, and the character at the position remains. The user prepared $\backslash[(_1.^*)_1\backslash].^*\backslash[\backslash1\backslash]$ to find substrings of the form $[x]y[x]$, where \[and \] match the characters [and], respectively, $.^*$ matches any string, and the backreference \1 refers to the string matched by the capturing group $(_1.^*)_1$. Additionally, to rule out substrings preceded by #, the regex checks the context using the negative lookbehind $(?{<}!\#)$. The user prepared \$1:\$2 to construct $x : y$ by referring to x and y of the matched substring $[x]y[x]$ using references \$1 and \$2.

Unfortunately, there are bugs in these user-prepared parameters. First, the user prepared the reference \$2 to refer to the substring matched to the second $.^*$, but forgot to write a capturing group whose index is 2. Therefore, the unassigned reference \$2 is treated as just the string \$2. Second, the second $.^*$ matches the substring "ab[a], [b]c" of the first text and not the intended substring "ab", due to the behavior of regex engines. Consequently, for the two texts, the outputs of replaceAll with the user-prepared parameters are "a:\$2" and "#[b]de[b], a:\$2", respectively, which are undesired.

Repair Problem. Our tool R2-DS can automatically repair the bugs by solving the replaceAll-repair problem. The repair problem takes a regex *reg*, a replacement *rep*, and a set $\mathcal{E}$ of *examples*, which reflect the user's intention about the repair. The solution is a pair of a regex *reg*′ and a replacement *rep*′ such that replaceAll using the pair as the parameters behaves consistently with the examples in $\mathcal{E}$ and the pair is syntactically close to the given pre-repair ones, or $\bot$, meaning that there is no solution for the given instance. The examples describe a desired input-output relation of the function to be synthesized. We write an example as $w_{in} \rightsquigarrow w_{out}$, where w_{in} and w_{out} are the input and output with *origin information*. Origin information is denoted by $\llparenthesis_i$ and $\rrparenthesis_i$, and $\llparenthesis_i w \rrparenthesis_i$ in the input (resp. output) indicates that w is the ith substring matched to the regex (resp. obtained from the replacement). For example, $\llparenthesis_1 a \rrparenthesis_1 a \llparenthesis_2 a \rrparenthesis_2 \rightsquigarrow \llparenthesis_1 X \rrparenthesis_1 a \llparenthesis_2 X \rrparenthesis_2$ means that the user wants to replace the input aaa with the output XaX by replacing the first and the last characters a with X. In the case of the motivating example, the user may prepare the examples "$\llparenthesis_1$[a]ab[a]$\rrparenthesis_1$, [b]c[a]$\rightsquigarrow\llparenthesis_1$a:ab$\rrparenthesis_1$, [b]c[a]" and "#[b]de[b], $\llparenthesis_1$[a]c[a]$\rrparenthesis_1\rightsquigarrow$#[b]de[b], $\llparenthesis_1$a:c$\rrparenthesis_1$". Note that, in PBE scenarios, the number of examples should be small for usability.

Repair Algorithm. R2-DS first checks whether the given instance has a solution, and if no then it outputs $\bot$ and halts immediately. Otherwise, it starts the repair process. The instance in our running example has a solution, and therefore, R2-DS moves on to the repair process. Roughly, the repair process iterates the following steps until it finds a solution: enumerating templates (*Searching Templates*) and checking whether the templates can be

instantiated into a solution using SMT constraint solving (*SMT Constraint Solving*).

Searching Templates. R2-DS enumerates pairs of *regex and replacement templates*. A regex (resp. replacement) template is a regex (resp. replacement) with *holes*, denoted by □. A hole is a placeholder to be replaced with a concrete expression. Intuitively, it represents a part that is currently under repair. To enumerate the templates, R2-DS changes the structures of regex and replacement templates by adding, reducing, and expanding holes. After some iterations, R2-DS finds the pair of the templates $((?<!\#)\backslash[(_1.^*)_1\backslash]\square^*\backslash[\backslash1\backslash], \$1:\square)$.

SMT Constraint Solving. R2-DS checks whether or not the template can be instantiated into a solution by replacing the holes in the regex template with a character set and replacing the holes in the replacement template with a character or a reference with insertions of capturing groups into a regex as needed. To check this, R2-DS prepares variables that correspond to the holes in the templates, and constructs an SMT constraint ϕ over the variables such that ϕ is satisfiable iff there is an instantiation of the regex and replacement templates that satisfy the examples. A subtle point here is that our constraints encode the information about insertions of capturing groups. We do this by inserting a *virtual* capturing group at each subexpression of the regex template to keep track of the substring matched by the subexpression, and refer to the information in the part of the constraint for replacement templates to decide whether we actually should insert the capturing groups.

Next, R2-DS checks the satisfiability of the constraint using an SMT solver. If satisfiable, it constructs the regex and the replacement pair from the satisfying assignment. Otherwise, it moves on to the *template pruning* to reduce the search space. Our pruning technique builds on the existing techniques for regex synthesis [12, 17, 18, 28, 35]; Namely, we build over- and under-approximations from the templates and check whether the approximations correctly classify the examples. However, we extend them to be *context-aware* (cf. Sec. 5.3 for details).

In the case of the above templates, the SMT constraint is satisfiable. From the satisfying assignment to the constraint, we obtain the regex $(?<!\#)\backslash[(_1.^*)_1\backslash](_2[\hat{}␣,\backslash[\backslash]]^*)_2\backslash[\backslash1\backslash]$ and the replacment \$1:\$2. Note that the regex template was instantiated by replacing its hole with the character set [^␣,\[\]] that accepts any character except for the white space ␣ and the characters ,, [, and], the replacement template was instantiated by replacing its hole with the new reference \$2, and a capturing group corresponding to the new reference was inserted at an appropriate subexpression of the regex.

3 PRELIMINARIES

This section briefly reviews the syntax and the semantics of regexes. Since there are several variants of regexes in practice [5, 13, 20], we focus on those of JavaScript, a popular programming language for building web applications with regex-dependent string functions. Specifically, this paper follows the semantics of regexes and regex-dependent string functions of the ECMAScript® 2023 Language Specification [22]. We first introduce the notations used in the paper.

General Notation. We write $\mathbb{N}$ for the set of natural numbers not including 0, $\mathbb{N}_0$ for $\mathbb{N} \cup \{0\}$, $[i..j]$ for the set $\{i, i+1, \cdots, j\}$ where $i, j \in \mathbb{N}$ and $i \leq j$, $[i]$ for $[1..i]$, and $\cdot$ for the concatenation of sequences. We will omit $\cdot$ if it is clear from context. For a sequence l, we write $|l|$ for its length, and for $i, j \in \{0, 1, \cdots, |l|-1\}$ with $i \leq j$, $l[i]$ for the ith element, $l[i..j]$ for $l[i] \cdot l[i+1] \cdots l[j]$, and $l[i..j)$ for $l[i..j-1]$ if $i \leq j-1$ and otherwise, i.e., if $i > j-1$, ϵ. We use _ to omit unimportant elements. For a function f, we use $dom(f)$ to denote the domain of f.

Regex. Let us fix a finite alphabet Σ. The syntax of regexes is defined:

$$\begin{aligned} r \quad ::= \quad & [C] \mid \epsilon \mid r \cdot r \mid r|r \mid r\{i,j\} \mid r\{i,j\}^? \mid \\ & (_k r)_k \mid \backslash k \mid (?=r) \mid (?!r) \mid (?<=r) \mid (?<!r) \end{aligned}$$

, where $C \subseteq \Sigma$, $i \in \mathbb{N}_0$, $j \in \mathbb{N}_0 \cup \{\infty\}$ with $i \leq j$, and $k \in \mathbb{N}$. Here, $i \leq \infty$ for all $i \in \mathbb{N}_0$. $[C]$ is a *character set*, which matches a character in C. For readability, we write a for $[\{a\}]$, $[ab\cdots c]$ for $[\{a, b, \cdots, c\}]$, . for $[\Sigma]$, and $[\hat{}C]$ for $[\Sigma \setminus C]$. The operators ϵ and $r \cdot r$ (or just rr when there is no ambiguity) are the empty string and the concatenation, respectively, and their semantics are standard. The operator $r_1|r_2$ is the *deterministic* union that first tries to match r_1, and if the whole matching cannot succeed with r_1, then it tries to match r_2. The operator $r\{i,j\}$ (resp. $r\{i,j\}^?$) is the *greedy* (resp. *lazy*) *bounded repetition*, which tries to match r at least i times and at most j times as many (resp. few) as possible if $j \in \mathbb{N}_0$. When $j = \infty$, it tries to match r at least i times as many (resp. few) as possible. From these operators, we can construct other quantifiers as syntactic sugar: $r^* = r\{0,\infty\}$, $r^{*?} = r\{0,\infty\}^?$, $r^+ = r\{1,\infty\}$, $r^{+?} = r\{1,\infty\}^?$, $r^? = r\{0,1\}$, and $r^{??} = r\{0,1\}^?$.

The remaining operators are real-world extensions. The operators $(_k r)_k$ and $\backslash k$ are the capturing group and the backreference, respectively. The capturing group $(_k r)_k$ tries to match r and, if the matching succeeds, it stores the matched substring into an *environment* with the index k. An environment Γ is a mapping from an index k to the substring matched to the kth capturing group. The backreference $\backslash k$ refers to the substring matched to the corresponding capturing group $(_k r)_k$. If there is no such substring (i.e., $k \notin dom(\Gamma)$), $\backslash k$ is evaluated to an empty string ϵ. Also, we assume that every capturing group in a regex has a unique index. That is, our semantics follows ϵ- and *no-label-repetition*-semantics [5], and this is actually consistent with the ECMAScript® Language Specification. The operators $(?=r)$ and $(?!r)$ are positive and negative lookaheads, respectively, which try to match r without any character consumption, such that $(?=r)$ (resp. $(?!r)$) succeeds if r succeeds (resp. fails). The operators $(?<=r)$ and $(?<!r)$ are positive and negative lookbehinds, respectively. They are analogous to lookaheads, but with the direction of the matching reversed, i.e., lookbehinds try to match from right to left.

We next briefly review the semantics of regexes. In this paper, we use the formal semantics defined by Chida and Terauchi [18] for repairing regexes that follows the ECMAScript® 2023 Language Specification. The semantics is defined as the deterministic matching relation $(r, w, p) \Downarrow x$, where $x \in \{(p', \Gamma'), \text{failed}\}$.[1] Roughly, this states that if $x = (p', \Gamma')$, then the matching of r on the input string w at the position p succeeds, changes the position p to p', and captures substring as described by the environment Γ'. If $x = \text{failed}$,

[1] We use a simplified version of the relation introduced in [18]. This corresponds to the judgement $(r, \epsilon, w, p, \emptyset, \text{forward}, \text{true}) \Downarrow _$, where ϵ is a continuation regex, $\emptyset$ is an environment, forward is a direction of the matching, and true is a flag, in [18].

it means that the matching fails at that state. We do not describe the rules of the relation in this paper and refer interested readers to [18] for full details. As pointed out in [38], the semantics introduced in [18] handles option operators $r^?$ and $r^{??}$ incorrectly. To ensure the correctness, we correct the bug by introducing bounded repetitions that were not considered in [18], extending the semantics of [18] with rules for them, and encoding options by them as mentioned above. The details can be found in the long version of this paper.

4 THE REPAIR PROBLEM

In this section, we define the replaceAll-repair problem. We first define in Sec. 4.1 *strings with origin information* and with them define *examples* for the problem. Next in Sec. 4.2, we define an *origin semantics* of replaceAll functions and formally define what it means for a regex and a replacement to be consistent with examples. In Sec. 4.3, we define the repair problem and show that it has unsolvable instances. Finally, we identify a necessary and sufficient condition for a problem instance to have a solution in Sec. 4.4.

4.1 String with Origin Information

A *string with origin information* is a pair (w, η) where w is a string and η is a list of pairs of integers of the form $[(p_1, p'_1), (p_2, p'_2), \cdots, (p_n, p'_n)]$ for some $n \in \mathbb{N}_0$ ($\eta = \epsilon$ if $n = 0$) satisfying $p_i, p'_i \in \mathbb{N}_0$ and $p_i \leq p'_i$ for all $i \in [n]$, $p'_j \leq p_{j+1}$ for all $j \in [n-1]$, and $p_i < p_j$ for all $i < j$ where $i, j \in [n]$. For a string with origin information (w, η), each element $\eta[i] = (p, p')$ of the origin information η indicates the substring $w[p..p']$. From this, we assume that $p' \leq |w|$ for each element. For readability, we use $(\!|_i \cdot |\!)_i$ to denote a substring represented by the ith element of η, i.e., $w[p..p']$ for (w, η) with $\eta[i] = (p, p')$. For example, we represent the string with origin information $(abcd, [(0, 1), (2, 3), (4, 4)])$ as $(\!|_1 a|\!)_1 b (\!|_2 c|\!)_2 d (\!|_3|\!)_3$.

An *example* is a tuple (w, η, θ), where (w, η) is a string with origin information and θ is *output fragments* which is a list of strings such that $|\eta| = |\theta|$. Here, w is an input to replaceAll functions, and η and θ describe how the output is obtained from the input. Specifically, an example (w, η, θ) states that regexes of desired replaceAll functions should match only the substrings $w[p, p']$ for each $\eta[i] = (p, p')$ where $0 \leq i < |\eta|$, and the matched substrings should be replaced with the output fragment $\theta[i]$. For instance, the example $(abac, [(0, 1), (2, 3)], [d, d])$ reflects the user's intention that for the given input $abac$, desired replaceAll functions should return the output $dbdc$ whose first and second characters d of the output originates from the first and second characters a of the input, respectively. Hereafter, we use $\text{output}(w, \eta, \theta)$ to denote the output obtained from w by replacing all substrings $w[p, p']$, where $\eta[i] = (p, p')$, with $\theta[i]$. Also, for readability, we use $w_{in} \rightsquigarrow w_{out}$ to denote an example (w, η, θ), where w_{in} is the string with origin information (w, η) and w_{out} is the string with origin information obtained from w by replacing all occurrences $(\!|_i w'|\!)_i$ of w_{in} with $(\!|_i \theta[i-1]|\!)_i$. For instance, we write the above example as $(\!|_1 a|\!)_1 b (\!|_2 a|\!)_2 c \rightsquigarrow (\!|_1 d|\!)_1 b (\!|_2 d|\!)_2 c$ and $\text{output}(abac, [(0, 1), (2, 3)], [d, d]) = dbdc$.

4.2 Origin Semantics

We first review the replaceAll function and then give its *origin semantics*. The replaceAll function takes a string w, a regex *reg*, and a *replacement rep*, and returns a string by replacing the substrings matched to *reg* with the strings constructed from *rep*. Replacements are defined by the following syntax: $s ::= \epsilon \mid a \mid \$i \mid s \cdot s$, where ϵ is the empty string, a is a character, $\$i$ is a *reference*, and $s \cdot s$ is a concatenation. A *reference* is of the form $\$i$ where $i \in \mathbb{N}$, and like a backreference of regexes, it refers to the substring associated with the index i in an environment Γ, i.e., $\Gamma(i)$. Hereafter, we refer to replaceAll(*reg*, *rep*) as replaceAll with the regex parameter *reg* and the replacement parameter *rep*.

$$\frac{p < |w| \quad (reg, w, p) \Downarrow (p', \Gamma) \quad [\![rep]\!]\,\Gamma = x \quad p'' = ite(p' \neq p, p', p+1) \quad (w, p'', reg, rep) \hookrightarrow (w', \eta)}{(w, p, reg, rep) \hookrightarrow (x \cdot w', [(p, p')] \cdot \eta)}$$

$$\frac{p = |w| \quad (reg, w, p) \Downarrow (p', \Gamma) \quad [\![rep]\!]\,\Gamma = x}{(w, p, reg, rep) \hookrightarrow (x, [(p, p')])}$$

$$\frac{p < |w| \quad (reg, w, p) \Downarrow \text{failed} \quad (w, p+1, reg, rep) \hookrightarrow (w', \eta)}{(w, p, reg, rep) \hookrightarrow (w[p] \cdot w', \eta)}$$

$$\frac{p = |w| \quad (reg, w, p) \Downarrow \text{failed}}{(w, p, reg, rep) \hookrightarrow (\epsilon, [])}$$

Figure 1: Rules for replaceAll**. The function *ite* is defined by: *ite*(true, A, B) = A and *ite*(false, A, B) = B.**

The origin semantics of replaceAll is defined as the relation $(w, p, reg, rep) \hookrightarrow (w', \eta)$, where w and w' are strings, p is a position in w, *reg* is a regex, *rep* is a replacement, and η is an origin information. Roughly, the relation $(w, p, reg, rep) \hookrightarrow (w', \eta)$ states that running replaceAll(*reg*, *rep*) on w from the position p outputs the string with origin information (w', η).

Figure 1 shows the rules defining the semantics. Here, we use the deterministic matching relation $(reg, w, p) \Downarrow _$ described in Sec. 3 to obtain the results of regex matching. $[\![rep]\!]\,\Gamma$ denotes the string constructed from the replacement *rep* under the environment Γ. Formally, it is defined by: $[\![\epsilon]\!]\,\Gamma = \epsilon$, $[\![a \cdot rep]\!]\,\Gamma = a \cdot ([\![rep]\!]\,\Gamma)$, and $[\![\$i \cdot rep]\!]\,\Gamma = \Gamma(i) \cdot ([\![rep]\!]\,\Gamma)$. Here, we let $\Gamma(i) = \epsilon$ if $i \notin dom(\Gamma)$.[2] There is a rule per each of the following four situations: the matching (reg, w, p) succeeds or not and the position is the end of the input string or not. If the matching succeeds, i.e., $(reg, w, p) \Downarrow (p', \Gamma)$, then the function appends the string $[\![rep]\!]\,\Gamma$ and the origin information (p, p') to the output, and continues the replacement from the position p' if $p \neq p'$ and $p+1$ if $p = p'$. If the matching fails, it appends the character c to the output, where $c = w[p]$ if $p < |w|$ and otherwise $c = \epsilon$, and continues the matching from the position $p+1$. If the position p reaches the end of w, i.e., $p = |w|$, it behaves similarly to the above rules, but the replacement ends.

Example 4.1. Consider the evaluation of replaceAll(a(?=$(_1.)_1$), $X\$1X$) on the input string ab. The derivation tree is as follows. Here, A_i denotes $(a(?=(_1.)_1), ab, i)$ for $i \in \{0, 1\}$.

[2] This differs from the actual behavior of replaceAll in ECMAScript. Namely, in the actual replaceAll, if a reference i is unassigned (i.e., $i \notin dom(\Gamma)$), the string constructed from $\$i$ depends on the regex. Specifically, if the number of capturing groups in the regex is less than i, $\$i$ is evaluated to the string $\$i$; otherwise, it is evaluated to ϵ. Although it is easy to modify the rules to follow the actual behavior, for simplicity, our rules always evaluate unassigned references to ϵ. Note that if one wants to write the string $\$i$ in a replacement, they can do so without using an unassigned reference by writing $\$\$i$ [22].

$$\dfrac{\dfrac{\cdots}{A_0 \Downarrow (1, \{(1, b)\})} \quad [\![X\$1X]\!]\,\{(1, b)\} = XbX \quad B}{(ab, 0, a(?=(_1.)_1), X\$1X) \hookrightarrow (XbXb, [(0, 1)])}$$

The subderivation B is as follows.

$$\dfrac{A_1 \Downarrow \text{failed} \quad \dfrac{\cdots}{(ab, 2, a(?=(_1.)_1), X\$1X) \hookrightarrow (\epsilon, [])}}{(ab, 1, a(?=(_1.)_1), X\$1X) \hookrightarrow (b, [])}$$

That is, for the input string ab, replaceAll($a(?=(_1.)_1), X\$1X$) replaces the substring a with the string XbX and leaving the other parts unchanged.

We are now ready to define the notion of *consistency with examples*.

Definition 4.2 (Consistency). We say that replaceAll(reg, rep) *is consistent with* (or, *satisfies*) an example (w, η, θ) if $(w, 0, reg, rep) \hookrightarrow (\text{output}(w, \eta, \theta), \eta)$. For a set of examples $\mathcal{E}$, we say that replaceAll(reg, rep) *satisfies* the set of examples if it satisfies all examples in $\mathcal{E}$.

For instance, replaceAll($a(?=(_1.)_1), X\$1X$) satisfies the example $(ab, [(0, 1)], [XbX])$ because $\text{output}(ab, [(0, 1)], [XbX]) = XbXb$ and $(ab, 0, a(?=(_1.)_1), X\$1X) \hookrightarrow (XbXb, [(0, 1)])$.

4.3 Problem Formulation

Before we define the repair problem, we consider a metric to measure the quality of a repair. We base our metric on those from the prior works for regex synthesis [17, 18, 35]. They used a notion called *edit distance* between two regexes so that a regex of a short distance from a pre-repair regex is deemed to be of high quality. The metric is justified by the assumption that the pre-repair regex may be incorrect but close to the intended one. Like the prior works, we deem parameters (i.e., a regex and a replacement) of a short distance from pre-repair parameters to be high quality. Since we consider repairing a replacement in addition to a regex, we use the sum of edit distances of the two as the metric, but with a certain modification on the definition of edit distance regarding insertions of capturing groups.

Definition 4.3 (Capture-insertion relation). Given a set of integers I, we define the relation $\Rightarrow_I$ over regexes where $r \Rightarrow_I r'$ iff it is possible to rewrite r to r' using associativity of the binary operators (i.e., the concatenation and the union) and inserting capturing groups $(_i\cdot)_i$ where $i \notin I$.

As explained below, the relation plays the role of inserting capturing groups into a regex at *no cost*. Let $|r|$ denote the number of AST nodes of r.

Definition 4.4 (Edit Distance). For subtrees $r_1, r_2, \cdots, r_n$ of a regex r, the *edit* $e = r[r'_1/r_1, \cdots, r'_n/r_n]$ replaces r_i with r'_i where $i \in [n]$. The *cost* of the edit is $\sum_{i\in[n]}, |r_i| + |r'_i|$. Then, for two regexes r_1 and r_2, the edit distance between r_1 and r_2 is the minimum cost of an edit that rewrites r_1 to some r'_2 s.t. $r'_2 \Rightarrow_I r_2$ where I is the set of indexes of capturing groups and backreferences in r'_2.

We lift the edit distance to replacements by interpreting a replacement as a regex consisting of a character set, a backreference, and a concatenation. Then, the edit distance between two parameters $f_1 = (reg_1, rep_1)$ and $f_2 = (reg_2, rep_2)$, denoted $D(f_1, f_2)$, is $d_{reg} + d_{rep}$, where d_{reg} (resp. d_{rep}) is the edit distance between reg_1 and reg_2 (resp. rep_1 and rep_2). For example, for $f_1 = (a \cdot b \cdot c, d \cdot e)$ and $f_2 = (e^* \cdot (_1 b \cdot c)_1, d \cdot f \cdot \$1)$, $D(f_1, f_2) = 7$ because the edit distance between the regexes is 3 (i.e., the cost of replacing a with e^*) and the edit distance between the replacements is 4 (i.e., the cost of replacing e with $f \cdot \$1$). Note that there is no cost to insert a capturing group into $b \cdot c$ because $e^* \cdot b \cdot c \Rightarrow_\emptyset e^* \cdot (_1 b \cdot c)_1$.

Note that edit distance used in prior works [17, 18, 35] is a restriction of Definition 4.4 obtained by asserting $r'_2 = r_2$, i.e., it is the minimum cost of an edit that rewrites r_1 to r_2. We extend the prior definition by the capture-insertion relation because insertions of capturing groups are quite common in practice (e.g., [39, 41]) but the prior definition allots high costs to such an insertion as it would be represented by a replacement of r to $(_i r)_i$ which costs $2\cdot|r|+1$. We refer to the long version of this paper for further details including real-world cases that motivated our new definition.

It is worth noting that we can compute the minimum edit distances efficiently by treating binary operators as n-ary operators and computing the normal form. For example, $\cdot(a, b, c)$ is the unique normal form of both $(a \cdot b) \cdot c$ and $a \cdot (b \cdot c)$.

We now define the repair problem.

Definition 4.5 (The Repair Problem). Given a regex reg, a replacement rep, and a set of examples $\mathcal{E}$, the replaceAll-*repair problem* is the problem of synthesizing a pair $f_2 = (reg', rep')$ of a regex and a replacement such that f_2 satisfies $\mathcal{E}$ and the distance between $f_1 = (reg, rep)$ and f_2 is minimal, i.e., for any $f_3 = (reg'', rep'')$ that satisfies $\mathcal{E}$, $D(f_1, f_2) \leq D(f_1, f_3)$.

Our repair problem is NP-hard. The proof is by a reduction from the *extraction-regex-repair* problem [18] that is known as NP-hard. We show the proof in the long version of this paper.

Theorem 4.6. *The repair problem is NP-hard.*

While a PBE regex repair or synthesis problem often has the guarantee that every instance has some solution [17, 18, 35], our problem has *unsolvable* instances.

Theorem 4.7. *The repair problem has an unsolvable instance.*

Proof. There is no pair of a regex and a replacement that satisfies the set of examples $\mathcal{E} = \{(\!|_1 a|\!)_1 \rightsquigarrow (\!|_1 b|\!)_1, (\!|_1 c|\!)_1 \rightsquigarrow (\!|_1 d|\!)_1\}$. To see this, note that the output fragments require that the replacement should be evaluated to the different characters b and d depending on the input string, and so, the replacement must have a reference. However, a reference can only refer to the characters that appear in the inputs, but these characters do not appear in the inputs. Therefore, there is no replacement that satisfies the examples. □

4.4 Necessary and Sufficient Condition

We now turn to identifying a necessary and sufficient condition for a problem instance to have a solution. We first introduce some notations. We use $out(\mathcal{E})$ to denote the set of all output fragments in $\mathcal{E}$, i.e., $out(\mathcal{E}) = \{w \in \Sigma^* \mid (_, _, \theta) \in \mathcal{E} \wedge \theta[i] = w \text{ for some } i\}$. Also, we define the function filter(w, w') that returns a string obtained from w by replacing all characters that appear in w' with ϵ. For example, filter($abcdeb, bd$) $= ace$. We say that v is a *scattered substring* of w if $w = w_1 \cdot v[0] \cdot w_2 \cdot v[1] \cdots w_m \cdot v[|v| - 1] \cdot w_{m+1}$. We now show the condition.

Theorem 4.8. *A repair problem instance $(reg, rep, \mathcal{E})$ is solvable if and only if the following holds: There exists a string $w \in \Sigma^*$ s.t. for*

Algorithm 1: The Repair Algorithm

Input: Regex reg, Replacement *rep*, Set of Examples $\mathcal{E}$
Output: Solution (reg', rep') or $\bot$ if there is no solution

1: **if** $\mathsf{isSolvable}(\mathcal{E}) = \mathsf{false}$ **then**
2:   **return** $\bot$
3: $Q \leftarrow \{(reg, rep)\}$
4: **while** Q is not empty **do**
5:   $(t_{reg}, t_{rep}) \leftarrow Q.pop()$
6:   **if** $\mathsf{isFeasible}(t_{reg}) = \mathsf{true} \land \mathsf{isFeasible}(t_{rep}) = \mathsf{true}$ **then**
7:     **if** $\mathsf{preTest}(t_{reg}) = \mathsf{true} \land \mathsf{preTest}(t_{rep}) = \mathsf{true}$ **then**
8:       $\phi \leftarrow \mathsf{genConst}(t_{reg}, t_{rep}, \mathcal{E})$
9:       **if** ϕ is satisfiable **then**
10:         $(reg', rep') \leftarrow \mathsf{completion}(t_{reg}, t_{rep}, \phi)$
11:         **return** (reg', rep')
12:     $Q.push(\mathsf{expandHole}(t_{reg}, t_{rep}))$
13:   $Q.push(\mathsf{addOrReduceHole}(t_{reg}, t_{rep}))$

all $w' \in out(\mathcal{E})$, w *is a scattered substring of* w' *and* $\mathsf{filter}(w', w_{in})$ *is a scattered substring of* w, *where* w_{in} *is the input string of* w'.

This condition rules out unsolvable instances. For instance, let us consider the set of examples $\{\llparenthesis_1 a \rrparenthesis_1 \rightsquigarrow \llparenthesis_1 ab \rrparenthesis_1, \llparenthesis_1 aa \rrparenthesis_1 \rightsquigarrow \llparenthesis_1 abb \rrparenthesis_1\}$ which does not satisfy the condition. To see that it is indeed unsolvable, note that from $\llparenthesis_1 aa \rrparenthesis_1 \rightsquigarrow \llparenthesis_1 abb \rrparenthesis_1$, we can see that the replacement of a solution must have bb in the subexpression. This is because the input aa does not contain b, and therefore, we cannot use a reference to construct bb. But then the output fragment of the other example (i.e., $\llparenthesis_1 ab \rrparenthesis_1$) must have a substring bb, which it does not.

The condition is also a sufficient condition. Roughly, this is shown by constructing a replacement that is consistent with all output fragments by using references, and a regex that matches correct substrings denoted by the origin information and extracts substrings that are required by the references in the replacement. We refer to the long version of this paper for the proof.

We emphasize that the condition is not only of theoretical interest but also useful in practice. For instance, let us consider the case where the user wants to replace all occurrences of `**x**` in a Markdown document with `<strong>x</strong>`, where x is a string, e.g., replacing "`**a** and **b**`" with "`<strong>a</strong> and <strong>b</strong>`". For this purpose, the user may prepare the example:

$\llparenthesis_1$**$\rrparenthesis_1$a$\llparenthesis_2$**$\rrparenthesis_2$ and $\llparenthesis_3$**$\rrparenthesis_3$b$\llparenthesis_4$**$\rrparenthesis_4$ $\rightsquigarrow$
$\llparenthesis_1$$\rrparenthesis_1a\llparenthesis_2$$\rrparenthesis_2$ and $\llparenthesis_3$$\rrparenthesis_3b\llparenthesis_4$$\rrparenthesis_4$.

While the example may at first seem intuitive, it turns out that there is no solution for this example. Indeed, the example violates the condition stipulated in Theorem 4.8, and automatically detected so by the algorithm given in Sec. 5.1. The user can then try to come up with a different example, e.g.,

$\llparenthesis_1$**a**$\rrparenthesis_1$ and $\llparenthesis_2$**b**$\rrparenthesis_2$ $\rightsquigarrow$
$\llparenthesis_1$a$\rrparenthesis_1$ and $\llparenthesis_2$b$\rrparenthesis_2$

which turns out to be solvable, and our tool is able to return a repair that is consistent with the example.

5 THE REPAIR ALGORITHM

Algorithm 1 overviews our repair algorithm. Our algorithm takes a repair problem instance $(reg, rep, \mathcal{E})$. It outputs either a pair (reg', rep') of a regex and a replacement that satisfies $\mathcal{E}$ and is minimal w.r.t. the edit distance from the input (reg, rep) or $\bot$, meaning that the given instance does not have a solution. Our algorithm consists of five steps: checking the necessary and sufficient condition (lines 1-2), initializing a template (line 3), pruning infeasible templates (lines 6-7), SMT constraint solving (lines 8-10), and template enumeration (lines 12 and 13). We next explain each step.

Checking Necessary and Sufficient Condition. Our algorithm first checks whether or not the given instance has a solution (line 1) by checking the condition given in Theorem 4.8. If the check fails, our algorithm immediately halts by outputting $\bot$ (line 2). Otherwise, it starts the repair process. The details are described in Sec. 5.1.

Initialize a Template. Our algorithm maintains a priority queue Q of pairs of a *regex template* and a *replacement template*. A regex (resp. replacement) template t_{reg} (resp. t_{rep}) is a regex (resp. replacement) with *hole* $\square$. A hole is a placeholder to be replaced with a concrete regex (resp. replacement). During the repair process, our algorithm adds/expands/reduces the holes to change the structure of the templates (cf. ***Template Enumeration***). As discussed in Sec 4.3, we treat the concatenation and the union as n-ary operators in the templates. Entries in Q are ordered according to the edit distance from the input regex and replacement pair in ascending order.[3] This ensures the minimality of repaired parameters. Initially, our algorithm pushes the input pair (reg, rep) into Q (line 3).

Pruning Infeasible Templates. Then, our algorithm retrieves the top element (t_{reg}, t_{rep}) of Q (line 5), and uses lightweight techniques to prune infeasible regex and replacement templates and reduce the search space (line 6). We first describe the pruning technique for regex templates. We define some terminologies. For $(w, \eta, \theta) \in \mathcal{E}$, we call the position i in w *positive* if there exists j s.t. $\eta[j] = (p, p')$ and $p \le i \le p'$, and *negative* otherwise. Our pruning technique checks for each $(w, \eta, \theta) \in \mathcal{E}$ and positive position p in w s.t. $\eta[i] = (p, p')$ (resp. negative position p), if t_{reg} cannot be instantiated into a regex that matches the substring $w[p, p')$ (resp. fails to match w at p). If so, we do not need to consider t_{reg} or the templates obtained by expanding the holes in t_{reg} and so can safely prune those templates. To check this, our technique constructs *context-aware* over- and under-approximations of t_{reg} and checks their matchings with the substrings at the positions. The pruning technique is implemented by the predicate $\mathsf{isFeasible}(t_{reg})$ whose details are deferred to Sec. 5.3.

As for pruning replacement templates, recall that a replacement can be seen as a regex composed of a character, a concatenation, and a backreference. Therefore, we build an over-approximation $r_\top$ from the replacement template by applying the methods from the prior work [17, 18, 35] and check whether the regex accepts all output fragments, i.e., we check $\forall w \in out(\mathcal{E}). w \in L(r_\top)$. If the check fails, then we safely prune the template and any templates obtainable by expanding the holes in it. The check is implemented by the predicate $\mathsf{isFeasible}(t_{rep})$.

[3] The edit distance between templates is defined in the same manner as that of regexes and replacements.

SMT Constraint Solving. If the templates (t_{reg}, t_{rep}) pass the pruning, then our algorithm checks whether they can be instantiated into a regex and replacement pair that satisfies the examples by replacing holes in the regex template with character sets and replacing holes in the replacement template with characters or references. To check this, our algorithm constructs an SMT constraint that is satisfiable iff there exists such an instantiation (line 8). The constraint contains the boolean variables that correspond to the holes in the templates, and they represent the ways of filling the corresponding holes with concrete expressions. Therefore, if the constraint is satisfiable, we can reconstruct the instantiation from the satisfying assignment. The details of the SMT constraint generation method are discussed in Sec. 5.2.

After building the constraint, our algorithm checks its satisfiability using an off-the-shelf SMT solver such as Z3 [21] (line 9). If it is satisfiable, our algorithm reconstructs a regex and replacement pair from t_{reg} and t_{rep} using the assignments (line 10) as remarked above, and applies the generalization method introduced in [35] that uses MaxSMT solving to the regex to enhance the quality of the repair. Namely, it tries to replace character sets that filled the holes with common character sets such as [a-z], [A-Z], and [0-9] while maintaining consistency with the examples. Note that the generalization process does not affect the edit distance.

Template Enumeration. If the templates (t_{reg}, t_{rep}) cannot be a solution, our algorithm changes their structures to generate additional templates. Specifically, our algorithm first expands a hole in the templates (line 12) by replacing it with an expression whose immediate subexpressions are holes. For example, let $t_{reg} = a|\square$ and $t_{rep} = \square \cdot b$. Then, $\mathsf{expandHole}(t_{reg}, t_{rep}) = \{(a|\epsilon, t_{rep}), (a|(\square \cdot \square), t_{rep}), \cdots, (a|(?<!\square), t_{rep})\} \cup \{(t_{reg}, \square \cdot \square \cdot b)\}$. Next, our algorithm adds a hole in the templates by replacing the leaf subexpression (i.e., a character set, ϵ, or a backreference for regex templates and a character or a reference for replacement templates) with a hole (line 13). Additionally, our algorithm reduces an expression whose immediate subexpressions are holes with a hole. For example, let $t_{reg} = \square^* \cdot a$ and $t_{rep} = \square \cdot \square \cdot b$. Then, $\mathsf{addOrReduceHole}(t_{reg}, t_{rep}) = \{(\square^* \cdot \square, t_{rep}), (\square \cdot a, t_{rep}), (t_{reg}, \square \cdot \square \cdot \square), (t_{reg}, \square \cdot b)\}$. This part is essentially the same as the prior works on regex synthesis and repair [17, 18, 35] with a straightforward adoption to replacement templates, and therefore, we refer to them for more details.

5.1 Checking the Condition

Recall that the condition shown in Theorem 4.8 requires that there exists a string $w \in \Sigma^*$ such that for all output fragments $w_{out} \in out(\mathcal{E})$, (i) w is a scattered substring of w_{out} and (ii) $\mathsf{filter}(w_{out}, w_{in})$ is a scattered substring of w, where w_{in} is the input string of w_{out}. We show an efficient method for checking the condition. Our key insight is that we can represent the strings that satisfy (i) and (ii) as pure regexes (i.e., regexes that do not have lookarounds and backreferences), and check the existence of such a string w by checking the non-emptiness of the intersection of these pure regexes. Precisely, in the case of (i), for each $w_{out,i} \in out(\mathcal{E})$, we can construct a regex that defines the set of all scattered substrings of $w_{out,i}$ as $r_i = w_{out,i}[0]^? \cdots w_{out,i}[|w_{out,i}| - 1]^?$. In the case of (ii), for each $w_{out,i} \in out(\mathcal{E})$ and the corresponding input string $w_{in,i}$, let $\mathsf{filter}(w_{out,i}, w_{in,i}) = w_{f,i}$. Then, we can construct a regex that defines the set of all strings that have $w_{f,i}$ as a scattered substring as $r'_i = .^* \cdot w_f[0] \cdots .^* \cdot w_f[|w_f| - 1] \cdot .^*$. Therefore, we can construct the set of strings that satisfies both (i) and (ii) as the intersection of these regexes, i.e., $L(r_1) \cap L(r_2) \cdots \cap L(r_{|out(\mathcal{E})|}) \cap L(r'_1) \cap L(r'_2) \cdots \cap L(r'_{|out(\mathcal{E})|})$. The condition is satisfied iff the intersection is not empty.

For example, $\mathcal{E} = \{(\!|_1 a|\!)_1 \rightsquigarrow (\!|_1 aab|\!)_1, (\!|_1 b|\!)_1 \rightsquigarrow (\!|_1 abb|\!)_1\}$. Here, $\mathsf{filter}(aab, a) = b$ and $\mathsf{filter}(abb, b) = a$. To check the condition, we construct the regexes described above as follows: $r_1 = a^? a^? b^?$, $r_2 = a^? b^? b^?$, $r'_1 = .^* b.^*$, and $r'_2 = .^* a.^*$. In this case, $ab \in L(r_1) \cap L(r_2) \cap L(r'_1) \cap L(r'_2)$. Therefore, the condition is satisfied and so the instance is solvable.

Algorithm 2: genConstForExample

Input: Regex Template t_{reg}, Replacement Template t_{rep}, Example $e = (w, \eta, \theta)$
Output: SMT Constraint ϕ

1: $\phi \leftarrow \bigwedge_{i \in [n]} (\Sigma_{b \in \mathfrak{B}_i} b = 1)$
2: **for** $p \in \{0, 1, \cdots, |w|\}$ **do**
3: $\quad S_{reg} \leftarrow \mathsf{compute}S_{reg}(t_{reg}, w, p)$
4: $\quad$ **if** p is positive and $\eta[j] = (p, p')$ for some j **then**
5: $\quad\quad \phi \leftarrow \phi \wedge \bigvee_{(p', \Gamma', \phi') \in S_{reg}} (\phi' \wedge \Phi_{rep}(t_{rep}, \theta[j], \Gamma'))$
6: $\quad$ **else if** p is negative **then**
7: $\quad\quad \phi \leftarrow \phi \wedge \neg(\bigvee_{(_, _, \phi') \in S_{reg}} \phi')$
8: **return** ϕ

5.2 SMT Constraint Generation

We describe the SMT constraint generation method. The SMT constraint contains the following boolean variables. For the regex template t_{reg}, we use the boolean variables v_i^a, such that v_i^a is true iff the character set $[C]$ that corresponds to the ith hole of t_{reg} accepts the character a, i.e., $a \in C$. In what follows, we assume that each node t of a regex template is identified by a unique index c_t. For the replacement template t_{rep}, we use three types of boolean variables: v_i^a, $v_i^{\$j}$, and $v_i^{\$c_t}$, where $a \in \Sigma$, $i, j \in \mathbb{N}$, and c_t is an index identifying a node of the AST of t_{reg} , such that v_i^a (resp. $v_i^{\$j}$) is true iff the ith hole of t_{rep} is a (resp. the reference $\$j$). The variables $v_i^{\$c_t}$ are used to insert new capturing groups. Specifically, $v_i^{\$c_t}$ is true iff the subexpression that corresponds to the ith hole of t_{rep} is the reference $\$c_t$ with the capturing group whose index is c_t and whose immediate subexpression is t newly inserted into t_{reg}. The insertion is done in a way that respects the n-ary operator normal form (cf. Sec. 4.3). That is, we regard $t = \odot(t_k, \cdots, t_l)$ to be a subexpression of $t' = \odot(t_1, t_2, \cdots, t_k, \cdots, t_l, \cdots, t_n)$ where $\odot \in \{\cdot, |\}$ so that the index c_t identifies t and inserting a capturing group at t changes t' to $\odot(t_1, t_2, \cdots, t_{k-1}, (_{c_t} \odot(t_k, \cdots, t_l))_{c_t}, t_{l+1}, \cdots, t_n)$ in the new template.

With these variables, we build the SMT constraint ϕ as follows: $\phi = \bigwedge_{e \in \mathcal{E}} \mathsf{genConstForExample}(t_{reg}, t_{rep}, e)$. That is, for each example $e \in \mathcal{E}$, we construct an SMT constraint that encodes the condition for satisfying e. Algorithm 2 shows genConstForExample. First, it initializes ϕ with the constraints asserting that for each hole in t_{rep}, the number of boolean variables corresponding to the hole

and assigned to true is exactly 1 (line 1). Here, n is the number of holes in t_{rep}, and $\mathfrak{B}_i$ is the set of boolean variables that correspond to the ith hole of t_{rep}. Then, for each position p s.t. $\exists j.\ \eta[j] = (p, _)$ or p is negative (cf. Sec. 5 ***Pruning Infeasible Templates***), it simulates the matchings of regexes instantiated from t_{reg} on w at p and computes the set S_{reg} of succeeded *matching results* (line 3). A matching result is of the form (p', Γ', ϕ') where p' is a position in w, Γ' is an environment, and ϕ' is an SMT constraint. The triple means that if t_{reg} is instantiated into a regex r according to a satisfying assignment to ϕ', the matching of r on w at position p moves the position to p' and the environment after the matching is Γ'. Using these triples, we construct the SMT constraint for the example at the position p as follows: if $\exists j.\eta[j] = (p, _)$, for each matching result $(p', \Gamma', \phi') \in S_{reg}$, we construct a constraint $\Phi_{rep}(t_{rep}, \theta[j], \Gamma')$ that encodes the condition that t_{rep} outputs the correct output fragment $\theta[j]$ under Γ' and take the conjunction of the constraint with ϕ' (lines 4-5). If the position is negative, we take the negation of the disjunction of all constraints for succeeding the matching (lines 6-7). The encoding of the regex and replacement templates at the position p is done by the functions computeS_{reg} (line 3) and Φ_{rep} (line 5), respectively. We elaborate on these functions in the rest of this section.

SMT Constraint Generation for Regex Templates. We adopt and extend the method introduced in [18]. Namely, like the method of [18], computeS_{reg} uses the following judgment to obtain the matching results: $(t, w, p) \dashrightarrow S_{reg}$, where t is a regex template, w is an input string, p is a position in w, and S_{reg} is the set of succeeded matching results.[4] However, to accommodate the insertion of new captures that can be referred from the replacement, we extend the method to keep track of substrings matched by subexpressions by insert a *virtual* capturing group to each subexpression of t_{reg}, except for the concatenation and the union (insertion of new capturing groups at concatenations and unions are handled by the SMT constraint for the replacement described below). Recall that each node t of the AST of a regex template has a unique index c_t. We use the indexes for inserting virtual capturing groups. For example, the result of inserting virtual capturing groups into ab^* is $({}_{c_a}a)_{c_a}({}_{c_{b^*}}({}_{c_b}b)^*_{c_b})_{c_{b^*}}$. Consequently, compute$S_{reg}(t_{reg}, w, p) = S_{reg}$, where $(t'_{reg}, w, p) \dashrightarrow S_{reg}$ and t'_{reg} is the regex template obtained from t_{reg} by inserting virtual capturing groups as described above.

SMT Constraint Generation for Replacement Templates. The function Φ_{rep} is defined as: $\Phi_{rep}(t_{rep}, w, \Gamma) = \phi_{rep} \wedge \neg\phi_{\mathsf{overlap}}$. Here, ϕ_{rep} is the SMT constraint that encodes the condition that t_{rep} is consistent with the output fragment w under the environment Γ and $\neg\phi_{\mathsf{overlap}}$ is the SMT constraint that rules out assignments that require inserting capturing groups into n-ary operators with an *overlap*. We elaborate on the construction of ϕ_{rep} and ϕ_{overlap} below.

First, ϕ_{rep} is defined as: $\phi_{rep} = \Phi'(t_{rep} \cdot \dashv, w \cdot \dashv, \Gamma)$, where $\dashv \notin \Sigma$ is a special *endmarker* character that denotes the end of a sequence. Φ' is defined by: $\Phi'(\dashv, w, \Gamma) =$ true if $w[0] = \dashv$ and false otherwise, $\Phi'(a \cdot t_{rep}, w, \Gamma) = \Phi'(t_{rep}, w[1..|w|), \Gamma)$ if $w[0] = a$ and false otherwise, $\Phi'(\$i \cdot t_{rep}, w, \Gamma) = \Phi'(t_{rep}, v, \Gamma)$ if $w = [\![\$i]\!]\,\Gamma \cdot v$ and false otherwise, and $\Phi'(\square_i \cdot t_{rep}, w, \Gamma) = \phi_a \vee \phi_{\$j} \vee \Phi_{\$c_t}(\square_i \cdot t_{rep}, w, \Gamma)$ where $\square_i$ denotes the ith hole in the replacement template passed to Φ_{rep}. In the last case, $\phi_a = \Phi'(w[0] \cdot t_{rep}, w, \Gamma) \wedge v_i^{w[0]}$ if $w[0] \neq \dashv$ and false otherwise, $\phi_{\$j} = \bigvee_{j \in dom(\Gamma)} (\Phi'(\$j \cdot t_{rep}, w, \Gamma) \wedge v_i^{\$j})$, and the function $\Phi_{\$c_t}$ is defined below.

Algorithm 3: $\Phi_{\$c_t}$

Input: Replacement Template $t_{rep} = \square_i \cdot t'_{rep}$, Output fragment w, Environment Γ
Output: SMT Constraint ϕ

1: $\phi \leftarrow$ false
2: **for all** $c_t \in dom(\Gamma)$ **do**
3: **if** $t = \odot(t_1, t_2, \cdots, t_n)$ where $\odot \in \{\cdot, |\}$ **then**
4: **for** p from 1 to n, p' from $p+1$ to n **do**
5: **if** $\exists u.w = \Gamma(c_{t_p}) \cdots \Gamma(c_{t_{p'}}) \cdot u$ **then**
6: $\phi \leftarrow \phi \vee (\Phi'(t'_{rep}, u, \Gamma) \wedge v_i^{\$c_{\odot(t_p \cdots t_{p'})}})$
7: **else if** $\exists u.w = \Gamma(c_t) \cdot u$ **then**
8: $\phi \leftarrow \phi \vee (\Phi'(t'_{rep}, u, \Gamma) \wedge v_i^{\$c_t})$
9: **return** ϕ

Roughly, Φ' evaluates t_{rep} on w from left to right and checks whether the string obtained from the first element t of $t_{rep} = t \cdot t'_{rep}$ is consistent with the prefix of w. Specifically, $\Phi'(t \cdot t_{rep}, w, \Gamma)$ checks whether $[\![t]\!]\,\Gamma$ is a prefix of w if $t \neq \square$ and otherwise, i.e., $t = \square$, it simulates all the possible ways of filling the hole, i.e., filling the hole with a character a, a reference $\$j$ s.t. $({}_j r)_j$ appears in t_{reg}, or a reference $\$c_t$ that requires wrapping the subexpression t with the capturing group whose index is c_t and refers to it. The first two cases are analogous to the cases of $\Phi'(a \cdot t_{rep}, w, \Gamma)$ and $\Phi'(\$j \cdot t_{rep}, w, \Gamma)$, respectively, but with the difference that we take the conjunction of the variable v_i^x, where $x \in \{w[0], \$j\}$, with the SMT constraint constructed from the remaining encoding. The third case tries to insert a new capturing group into a regex and refers to it. This is accomplished by the function $\Phi_{\$c_t}$ shown in Algorithm 3. The basic idea is similar to the case for $\$j$, except for the handling of n-ary operators. Namely, for the case that the virtual capturing group under consideration is an n-ary concatenation or union, we simulate all possible insertions of a capturing group into the subexpressions. That is, for an n-ary operator $t = \odot(t_1, \cdots, t_n)$ and each of its subexpression $t' = \odot(t_p, \cdots, t_{p'})$ where $1 \leq p < p' \leq n$, we simulate the insertion $\odot(t_1, \cdots, t_{p-1}, ({}_{c_{t'}}\odot(t_p, \cdots, t_{p'}))_{c_{t'}}, t_{p'+1}, \cdots, t_n)$ (lines 3-6).

Secondly, ϕ_{overlap} is defined as: $\phi_{\mathsf{overlap}} = \bigvee_{(v_1, v_2) \in V} (v_1 \wedge v_2)$, where V is the set of pairs $(v_i^{\$c_t}, v_j^{\$c_{t'}})$ of variables s.t. the subexpressions t and t' has an overlap. Precisely, $(v_i^{\$c_t}, v_j^{\$c_{t'}}) \in V$ if and only if (1) $v_i^{\$c_t}$ and $v_j^{\$c_{t'}}$ both appear in ϕ_{rep}, (2) neither t is a subtree of t' nor t' is a subtree of t, and (3) t and t' share a common node. For example, let us consider the regex template abc, the replacement template $\square_1\square_2$, and the output fragment $abbc$. In this case, the replacement $\$c1\$c2$ instantiated from $\square_1\square_2$ can satisfy the output fragment $abbc$ if the references $\$c1$ and $\$c2$ refer to the subexpressions ab and bc of the regex template, respectively. However, if we insert the capturing groups in such a way, then it yields an overlap of capturing groups, i.e., it yields the expression

[4] This corresponds to $(t, \epsilon, w, p, \emptyset, \text{forward}, \text{true}) \dashrightarrow (S_{reg}, _)$, where ϵ is the initial continuation regex template, $\emptyset$ is the initial environment, forward is the direction, true is the flag, in the actual rules given in [18].

$(_{c1}a(_{c2}b)_{c1}c)_{c2}$ which is invalid according to the syntax of regexes. Therefore, to rule out such invalid insertions, we add the SMT constraint $\neg\phi_{\text{overlap}} = \neg(v_1^{\$c_{ab}} \wedge v_2^{\$c_{bc}})$.

5.3 Pruning Infeasible Templates

We show the construction of over- and under-approximations of a regex template for substring matching. The construction is composed of two steps: *approximation* and *addition of the context*. The first step uses the existing approximations of regex templates for whole matching [17, 18, 28, 35]. Roughly, the existing methods replace holes with $.^*$ or $\emptyset$, depending on the polarity. For example, let us consider the regex template $(?<=a\square)\square(?!\square)$. The over- and under-approximations are $(?<=a.^*).^*(?!\emptyset)$ and $(?<=a\emptyset)\emptyset(?!.^*)$, respectively. The second step extends the approximations to be context-aware.

We first consider the over-approximation. For a position p s.t. $\eta[i] = (p, p')$ for some i and $(w, \eta, \theta) \in \mathcal{E}$, we prepend $\hat{}\,w[0..p)$ and append $w[p'..|w|)\$$ to the over-approximation, where ^ and \$ are *anchors* that match the start and end of a string, respectively, and are syntactic sugar of lookarounds [26]. Specifically, let $r_\top$ be the over-approximation. Then, the context-aware over-approximation is $r'_\top = \hat{}\,w[0..p)\cdot r_\top \cdot w[p'..|w|)\$$, and we implement isFeasible(t_{reg}) to reurn false if $w \notin L(r'_\top)$. For example, let us consider the regex template $(?<!\square|a)b$ and the example $a(\!|_1 b|\!)_1 c \rightsquigarrow _$. Then, the context-aware over-approximation $r'_\top$ is $\hat{}\,a\cdot(?<!\emptyset|a)b\cdot c\$$. Since $abc \notin L(r'_\top)$, we can safely avoid expanding the hole. Indeed, we cannot obtain a solution from the regex template regardless of the replacement template because the negative lookbehind always fails to match the substring b.

We next consider making the under-approximation context-aware. This is similar to the case of over-approximation, but in this case, there is no need to care about the position where the matching finishes. Therefore, we append $.^*\$$ to the under-approximation. Specifically, let $r_\bot$ be the under-approximation. Then, for each negative position p, the context-aware under-approximation is $r'_\bot = \hat{}\,w[0..p)\cdot r_\bot \cdot .^*\$$, and we implement isFeasible(t_{reg}) to return false if $w \in L(r'_\bot)$. For example, let us consider the regex template $(?<!d\square)$. and the example $a(\!|_1 b|\!)_1 c \rightsquigarrow _$. The context-aware under-approximation at position 2 is $\hat{}\,ab\cdot(?<!d.^*).\cdot.^*\$$. We make isFeasible(t_{reg}) return true if neither cases apply.

Finally, like the prior works [17, 18, 35], we use a variant preTest of the pruning techniques (line 7). The variant is almost the same as the over-approximation case of isFeasible except that we replace holes with . instead of $.^*$.

6 OTHER REGEX-DEPENDENT STRING FUNCTIONS

We discuss how to apply our core ideas to other representative regex-dependent string functions such as search, replace, match, and exec.

Like replaceAll, replace takes a string as the input, and a regex and replacement pair as the parameter, and returns an output string. The only difference is that, whereas replaceAll replaces all matched substrings, replace replaces only the *first* matched substring. We can adapt our algorithm to repair the parameters of replace by a minor modification to the pruning techniques and the SMT constraint generation method for regex templates so that they stop after reaching the only position denoted by the origin information.

The search function takes a string as the input and a regex as the parameter, and returns the first matched position or -1 if there are no substrings of the string matching the regex. We can treat the repair problem for search as a special case of the repair problem for replace. That is, we add an origin information to the position of the example string where the substring should be matched (or with no origin information for an example string on which the to-be-synthesized function should return -1) and set the output fragments for all examples and the replacement parameter to be some fixed character $a \in \Sigma$ (our algorithm is optimized to not modify the replacement in such a case).

The match function takes a string as the input and a regex as the parameter, and returns a list of matched substrings. If the match function has the global flag, it returns a list of all matched substrings; otherwise, it returns a list containing only the first matched substring. We can treat this in the same way as that of search described above if it does not have a global flag. If it has, it can be handled as replaceAll by letting the origin information identify the positions of substrings that should be matched, and setting the output fragments and the replacement parameter to a fixed character.

The exec function takes a string as the input and a regex as the parameter, and returns a list whose first element is the first matched substring and the other elements are substrings extracted by capturing groups. From this, it can be handled in a similar way as the approach for search with a minor extension. That is, to address the substrings to be extracted, we add information about the substrings to be extracted into examples and extend the algorithm to check the correctness of the extraction. The extension to the algorithm is only a minor one because, as described in Sec. 5.2, the SMT constraint generation method computes matching results that already include the information about extraction (i.e., the environment) to handle backreferences in regexes and references in replacements.

7 EVALUATION

We implemented our algorithm as a tool called R2-DS. It is written in Java and uses Z3 [21] as the SMT solver. We evaluated R2-DS to answer the following questions: (**RQ1**) Can R2-DS repair real-world bugs of regex-dependent string functions in a reasonable time? (**RQ2**) Are the repaired functions high-quality? (**RQ3**) How impactful are the newly introduced techniques of context-aware pruning and SMT-constraint-based insertion of captures? Our experiments were conducted on Intel(R) Xeon(R) Platinum 8360Y CPU @ 2.40GHz. To answer **RQ3**, we prepared R2-DS$_{wo/p}$ and R2-DS$_{wo/c}$, which are variants of R2-DS with the pruning technique and the insertion of capturing groups disabled, respectively.

Benchmarks. We collected our benchmark tasks from GitHub commits and StackOverflow posts. Specifically, we collected the parameters of before and after repairs as the incorrect and correct parameters. We prepared the examples by using the ones provided in the commits/posts when they are available, and adding new ones or removing inappropriate ones manually by ourselves as needed

Table 1: Solved instances.

	instances	running time (seconds)		
	Solved (120)	**min**	**med**	**max**
R2-DS	109 (90.8%)	0.117	0.763	1451.92
R2-DS$_{wo/p}$	90 (75%)	0.081	1.853	1795.3
R2-DS$_{wo/c}$	76 (63.3%)	0.145	0.853	681.805

(e.g., when none was provided or ones that trigger Regular Expression Denial-of-Service (ReDoS) vulnerability [19], which makes the regex matching slow, were provided). As a result, for each instance, we prepared at least 5 examples that have non-empty origin information (we call them *positive* examples), at least 5 examples that have empty origin information (we call them *negative* examples), and at most 15 examples in total.[5] From the examples, we used at most 5 examples for the synthesis. The remaining examples are used as *left-out* example sets to measure the quality of the repairs.

To find the commits and posts, we first looked at all commits in the lists provided by [23] that studied string-related bugs in a wide variety of GitHub projects written in JavaScript and [44] that studied regex-related bugs in popular GitHub projects such as Apache, Mozilla, Facebook, and Google. Among these, we have collected the commits that change the parameters of replaceAll, replace, or search.[6] This gave us 34 benchmarks. Next, we searched on GitHub Advanced Search and StackOverflow with keywords such as "replaceAll", "bug", and "JavaScript" and collected the ones that change the parameters of replaceAll, replace, or search. In total, we collected 120 benchmark tasks (21 from [23], 13 from [44], and 86 from GitHub Advanced Search and StackOverflow). Among the 120, 97 are instances of replaceAll, 15 are instances of replace, and 8 are instances of search. The average and maximum sizes of the regexes (resp. replacements), measured as the number of AST nodes, are 16.43 and 193 (resp. 7.06 and 105). Also, 5 regexes have a bounded repetition, 42 (resp. 4) regexes have a (resp. lazy) Kleene star, 15 (resp. 2) regexes have a (resp. lazy) Kleene plus, 12 regexes have an option, a regex has a backreference, 4 (resp. 1) regexes have a negative lookbehind (resp. lookahead), 20 (resp. 14) regexes have the anchor ˆ (resp. $) [26], and 20 replacements have a reference. The data set is provided as supplementary material.

Validation. Our algorithm generates only correct parameters, i.e., the function using the repaired parameters behaves consistently with the examples. To verify this, we prepared a JavaScript program and confirmed that the function with the repaired parameters indeed returns correct outputs for all inputs in all cases.

RQ1: Efficiency To answer **RQ1**, we measured the minimum, median, and maximum running times of solving the instances, with a timeout of 30 minutes.[7] Table 1 reports the results. R2-DS could solve 90.8% of instances (91.7% of replaceAll instances, 80% of replace instances, and 100% of search instances) within 50.71 seconds on average. In some cases, R2-DS could not find a solution within the time limit. We inspected these cases and found that, as the existing methods for regex synthesis [17, 18, 28, 35], R2-DS suffered from the cases that required large changes from the original. For example, R2-DS could not the repair of the regex [␣]+, where ␣ is the white space, when the intended solution was [␣]+(?=([^']*'[^']*')*[^']*$) because such a repair requires appending a large expression. In summary, *R2-DS can repair real-world regex-dependent functions efficiently.*

Additionally, we evaluated how R2-DS scales as the size of the regex-dependent string function parameters increases, by comparing the running times of R2-DS over the sizes. Figure 2 shows the plot. The points on the border colored in blue show that R2-DS could not find a solution within the time limit. As the figure shows, R2-DS could find a solution in a short time, even for large parameters. In summary, *the performance of R2-DS scales with respect to the size of regex-dependent string functions.*

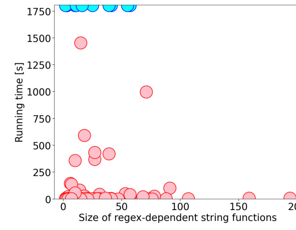

Figure 2: Scalability.

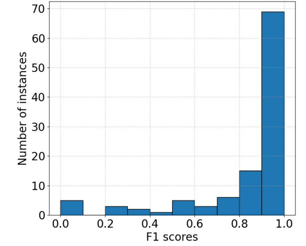

Figure 3: F1 scores.

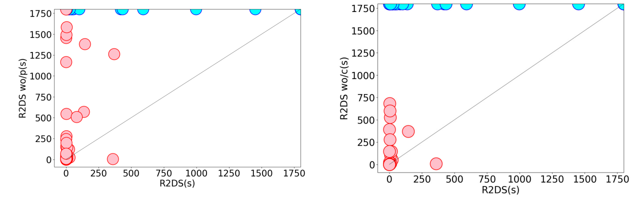

Figure 4: Impact of the pruning (left) and the capture insertions (right).

RQ2: Quality Following the approach used in [18, 35], we measured the quality of the repairs with respect to left-out example sets. That is, we measured the F1 scores using the left-out examples. The F1 score is calculated as $F1 = \frac{2 \times P \times R}{P+R}$, where P is the *precision* and R is the *recall*. The precision P is calculated as $P = \frac{TP}{TP+FP}$, where TP (resp. FP) is the number of positive (resp. negative) examples that are (resp. not) satisfied by the functions. The recall R is calculated as $R = \frac{TP}{TP+FN}$, where FN is the number of positive examples that are not satisfied by the functions. Therefore, the F1 score is between 0 and 1, and scores that are close to 1 imply the high quality of the repairs. Figure 3 shows the results. The median and average of the F1 scores are 1 and 0.84, respectively. We inspected the repairs with low F1 scores and observed that R2-DS could not find a high-quality repair when the instances require a large change from the original. In summary, *R2-DS can produce repairs that generalize well to left-out example sets, and therefore of high-quality.*

RQ3: Impact of the new techniques To evaluate the impact of the new techniques on R2-DS's overall performance, we conducted

[5] In some cases, the correct regex is one that accepts any strings, and so we do not prepare any negative examples for such cases.

[6] Due to time constraints, we do not evaluate R2-DS on match and exec. We leave them for future work.

[7] On an instance that has a solution, R2-DS is guaranteed to terminate and return a solution if given unbounded time and resource.

an ablation study in which we disabled some of them. In this evaluation, we consider the two ablations: R2-DS without the pruning techniques and R2-DS without insertions of capturing groups as SMT constraints, which we refer to as R2-DS$_{wo/p}$ and R2-DS$_{wo/c}$, respectively. We compared them against R2-DS, which incorporates both techniques.

Figure 4 shows the results. The points above the diagonal indicate that R2-DS could find a solution faster with the new technique rather than without it. The points colored in blue on the border indicate that the tool could not find a solution within the time limit. Overall, R2-DS achieved 85.13× (resp. 35.01%×) speedups against R2-DS$_{wo/p}$ (resp. R2-DS$_{wo/c}$) on average. We also observed that R2-DS$_{wo/p}$ (resp. R2-DS$_{wo/c}$) could not solve 19 (resp. 33) instances that R2-DS was able to solve. However, in a few cases, we observed negative performance impacts. We inspected such cases and observed that the new pruning technique can occasionally introduce ReDoS (a similar issue was observed in [18] for regex synthesis). As for the capture-insertion technique, the negative effect was also due to it changing edit distance (cf. Sec. 4.3) between templates that in some rare cases adversely affect their order in the priority queue. In summary, *the new pruning technique and constraint-based capturing-group-insertion technique introduced in the paper are highly impactful to the performance of the algorithm.*

8 RELATED WORK

There is a lot of work on synthesizing regexes from examples [1, 7–10, 12, 17, 18, 24, 25, 27, 28, 30, 35, 37, 42, 45]. However, all of the works focus only on regexes and do not consider replacements, which are essential for regex-dependent string functions like replaceAll and replace. Additionally, all of them focus on whole string matching and do not support substring matching, which is also essential for many regex-dependent string functions. In fact, as we showed in Sec. 4.4, our problem differs significantly from the problems addressed in the prior works in that, our problem has instances that do not have a solution, whereas the PBE regex synthesis and repair problems addressed in the prior works always have a solution. We mention that there are several methods for generating regexes based on genetic programming [2–4] that consider substring matching. However, they focus only on regexes and cannot be synthesize or repair regex-dependent string functions like replaceAll and replace that require replacements. Also, unlike ours or other PBE-based methods, these methods do not guarantee that the synthesis result is consistent with the given examples.

Origin information was introduced by Bojańczyk [6] to clarify how the output is constructed from the input in transducers. Inspired by the idea, we incorporated origin information into our examples to specify the desired outputs in the PBE scenario. To our knowledge, our work is the first to use origin information in the setting of PBE-based synthesis or repair.

Finally, we mention that there are many works on detecting bugs of regex-dependent string functions. Many of them focus on the replaceAll function due to its practical importance [13–16]. However, these works focus on detecting bugs and do not consider synthesis or repair.

9 CONCLUSION

This paper presented the *first* PBE-based method for repairing regex-dependent string functions. Our algorithm builds on the techniques from the prior work on PBE-based regex sythesis and repair but incorporates the following key novelties to address the large search space: origin information in examples to identify the locations of substrings to be matched, a new substring-context-aware pruning technique, and a novel use of SMT constraints to insert captures that can referred from the replacement. Additionally, we identified a novel necessary and sufficient condition that can be used to detect and filter unrepairable instances. We implemented our algorithm as a tool called R2-DS and evaluated it on real-world benchmarks. The results show that R2-DS can repair real-world bugs efficiently, find high-quality repairs, and the new pruning and capture insertion techniques significantly improve the performance.

ACKNOWLEDGMENTS

We thank anonymous reviewers for their useful comments. This work was supported by JSPS KAKENHI Grant Numbers JP23K24826, JP20K20625, and JP23K20380.

DATA-AVAILABILITY STATEMENT

The artifact of this paper is publicly available at [36].

REFERENCES

[1] Dana Angluin. 1978. On the complexity of minimum inference of regular sets. *Information and Control* 39, 3 (1978), 337–350. https://doi.org/10.1016/S0019-9958(78)90683-6

[2] A. Bartoli, G. Davanzo, A. De Lorenzo, E. Medvet, and E. Sorio. 2014. Automatic Synthesis of Regular Expressions from Examples. *Computer* 47, 12 (dec 2014), 72–80. https://doi.org/10.1109/MC.2014.344

[3] Alberto Bartoli, Andrea De Lorenzo, Eric Medvet, and Fabiano Tarlao. 2014. Playing Regex Golf with Genetic Programming. In *Proceedings of the 2014 Annual Conference on Genetic and Evolutionary Computation* (Vancouver, BC, Canada) *(GECCO '14)*. Association for Computing Machinery, New York, NY, USA, 1063–1070. https://doi.org/10.1145/2576768.2598333

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

[5] Martin Berglund and Brink van der Merwe. 2022. Re-examining regular expressions with backreferences. *Theoretical Computer Science* (2022). https://doi.org/10.1016/j.tcs.2022.10.041

[6] Mikołaj Bojańczyk. 2014. Transducers with Origin Information. In *Automata, Languages, and Programming*, Javier Esparza, Pierre Fraigniaud, Thore Husfeldt, and Elias Koutsoupias (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 26–37.

[7] Alvis Brādzma. 1993. Efficient Identification of Regular Expressions from Representative Examples. In *Proceedings of the Sixth Annual Conference on Computational Learning Theory* (Santa Cruz, California, USA) *(COLT '93)*. Association for Computing Machinery, New York, NY, USA, 236–242. https://doi.org/10.1145/168304.168340

[8] Alvis Brāzma. 1995. Learning of regular expressions by pattern matching. In *Computational Learning Theory*, Paul Vitányi (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 392–403.

[9] Alvis Brāzma and Kārlis Čerāns. 1994. Efficient learning of regular expressions from good examples. In *Algorithmic Learning Theory*, Setsuo Arikawa and Klaus P. Jantke (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 76–90.

[10] Alvis Brāzma and Kārlis Čerāns. 1997. Noise-tolerant efficient inductive synthesis of regular expressions from good examples. *New Gen. Comput.* 15, 1 (mar 1997), 105–140. https://doi.org/10.1007/BF03037562

[11] Carl Chapman and Kathryn T. Stolee. 2016. Exploring Regular Expression Usage and Context in Python. In *Proceedings of the 25th International Symposium on Software Testing and Analysis* (Saarbrücken, Germany) *(ISSTA 2016)*. Association for Computing Machinery, New York, NY, USA, 282–293. https://doi.org/10.1145/2931037.2931073

[12] Qiaochu Chen, Xinyu Wang, Xi Ye, Greg Durrett, and Isil Dillig. 2020. Multi-Modal Synthesis of Regular Expressions. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation* (London, UK) *(PLDI 2020)*. Association for Computing Machinery, New York, NY, USA, 487–502. https://doi.org/10.1145/3385412.3385988

[13] Taolue Chen, Yan Chen, Matthew Hague, Anthony W. Lin, and Zhilin Wu. 2017. What is Decidable about String Constraints with the ReplaceAll Function. 2, POPL, Article 3 (dec 2017), 29 pages. https://doi.org/10.1145/3158091

[14] Taolue Chen, Alejandro Flores-Lamas, Matthew Hague, Zhilei Han, Denghang Hu, Shuanglong Kan, Anthony W. Lin, Philipp Rümmer, and Zhilin Wu. 2022. Solving String Constraints with Regex-Dependent Functions through Transducers with Priorities and Variables. *Proc. ACM Program. Lang.* 6, POPL, Article 45 (jan 2022), 31 pages. https://doi.org/10.1145/3498707

[15] Taolue Chen, Matthew Hague, Jinlong He, Denghang Hu, Anthony Widjaja Lin, Philipp Rümmer, and Zhilin Wu. 2020. A Decision Procedure for Path Feasibility of String Manipulating Programs with Integer Data Type. In *Automated Technology for Verification and Analysis*, Dang Van Hung and Oleg Sokolsky (Eds.). Springer International Publishing, Cham, 325–342.

[16] Taolue Chen, Matthew Hague, Anthony W. Lin, Philipp Rümmer, and Zhilin Wu. 2019. Decision procedures for path feasibility of string-manipulating programs with complex operations. *Proc. ACM Program. Lang.* 3, POPL, Article 49 (jan 2019), 30 pages. https://doi.org/10.1145/3290362

[17] Nariyoshi Chida and Tachio Terauchi. 2022. Repairing DoS Vulnerability of Real-World Regexes. In *2022 IEEE Symposium on Security and Privacy (SP)*. 2060–2077. https://doi.org/10.1109/SP46214.2022.9833597

[18] Nariyoshi Chida and Tachio Terauchi. 2023. Repairing Regular Expressions for Extraction. *Proc. ACM Program. Lang.* 7, PLDI, Article 173 (jun 2023), 24 pages. https://doi.org/10.1145/3591287

[19] Scott Crosby. 2003. Denial of Service through Regular Expressions. USENIX Association, Washington, D.C.

[20] James C. Davis, Louis G. Michael IV, Christy A. Coghlan, Francisco Servant, and Dongyoon Lee. 2019. Why Aren't Regular Expressions a Lingua Franca? An Empirical Study on the Re-Use and Portability of Regular Expressions. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Tallinn, Estonia) *(ESEC/FSE 2019)*. Association for Computing Machinery, New York, NY, USA, 443–454. https://doi.org/10.1145/3338906.3338909

[21] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems* (Budapest, Hungary) *(TACAS'08/ETAPS'08)*. Springer-Verlag, Berlin, Heidelberg, 337–340.

[22] ECMA International. 2022. ECMAScript® 2023 Language Specification. https://tc39.es/ecma262/multipage/#sec-intro.

[23] Aryaz Eghbali and Michael Pradel. 2021. No strings attached: an empirical study of string-related software bugs. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering* (Virtual Event, Australia) *(ASE '20)*. Association for Computing Machinery, New York, NY, USA, 956–967. https://doi.org/10.1145/3324884.3416576

[24] Henning Fernau. 2009. Algorithms for learning regular expressions from positive data. *Information and Computation* 207, 4 (2009), 521–541. https://doi.org/10.1016/j.ic.2008.12.008

[25] Margarida Ferreira, Miguel Terra-Neves, Miguel Ventura, Inês Lynce, and Ruben Martins. 2021. FOREST: An Interactive Multi-tree Synthesizer for Regular Expressions. In *Tools and Algorithms for the Construction and Analysis of Systems*, Jan Friso Groote and Kim Guldstrand Larsen (Eds.). Springer International Publishing, Cham, 152–169.

[26] Jeffrey Friedl. 2006. *Mastering Regular Expressions.* O'Reilly Media, Inc.

[27] Efim Kinber. 2010. Learning Regular Expressions from Representative Examples and Membership Queries. In *Grammatical Inference: Theoretical Results and Applications*, José M. Sempere and Pedro García (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 94–108.

[28] Mina Lee, Sunbeom So, and Hakjoo Oh. 2016. Synthesizing Regular Expressions from Examples for Introductory Automata Assignments. *SIGPLAN Not.* 52, 3 (oct 2016), 70–80. https://doi.org/10.1145/3093335.2993244

[29] Yunyao Li, Rajasekar Krishnamurthy, Sriram Raghavan, Shivakumar Vaithyanathan, and H. V. Jagadish. 2008. Regular Expression Learning for Information Extraction. In *Proceedings of the 2008 Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics, Honolulu, Hawaii, 21–30. https://aclanthology.org/D08-1003

[30] Yeting Li, Shuaimin Li, Zhiwu Xu, Jialun Cao, Zixuan Chen, Yun Hu, Haiming Chen, and Shing-Chi Cheung. 2021. TransRegex: Multi-Modal Regular Expression Synthesis by Generate-and-Repair. In *Proceedings of the 43rd International Conference on Software Engineering* (Madrid, Spain) *(ICSE '21)*. IEEE Press, 1210–1222. https://doi.org/10.1109/ICSE43902.2021.00111

[31] Matthew Luckie, Bradley Huffaker, and k claffy. 2019. Learning Regexes to Extract Router Names from Hostnames. In *Proceedings of the Internet Measurement Conference* (Amsterdam, Netherlands) *(IMC '19)*. Association for Computing Machinery, New York, NY, USA, 337–350. https://doi.org/10.1145/3355369.3355589

[32] Louis G. Michael, James Donohue, James C. Davis, Dongyoon Lee, and Francisco Servant. 2019. Regexes Are Hard: Decision-Making, Difficulties, and Risks in Programming Regular Expressions. In *Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering* (San Diego, California) *(ASE '19)*. IEEE Press, 415–426. https://doi.org/10.1109/ASE.2019.00047

[33] Chris O'Hara. 2024. Validator.js. https://github.com/validatorjs/validator.js/ [Online; accessed 1-June-2024].

[34] OWASP. 2024. Input Validation Cheat Sheet. https://cheatsheetseries.owasp.org/cheatsheets/Input_Validation_Cheat_Sheet.html [Online; accessed 1-June-2024].

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

[36] R2-DS. 2024. https://github.com/NariyoshiChida/ASE2024/.

[37] 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.

[38] Noé De Santo, Aurèle Barrière, and Clément Pit-Claudel. 2024. A Coq Mechanization of JavaScript Regular Expression Semantics. arXiv:2403.11919 [cs.PL]

[39] StackOverflow. 2012. Javascript replace() and $1 issue. https://stackoverflow.com/questions/13368906/javascript-replace-and-1-issue?rq=3 [Online; accessed 1-June-2024].

[40] StackOverflow. 2013. replace in java with a regex doesn't replace from left to right. https://stackoverflow.com/questions/18901525/replace-in-java-with-a-regex-doesnt-replace-from-left-to-right [Online; accessed 1-June-2024].

[41] StackOverflow. 2017. Java regex not replacing my pattern. https://stackoverflow.com/questions/42778804/java-regex-not-replacing-my-pattern/42778862 [Online; accessed 1-June-2024].

[42] Mojtaba Valizadeh and Martin Berger. 2023. Search-Based Regular Expression Inference on a GPU. *Proc. ACM Program. Lang.* 7, PLDI, Article 160 (jun 2023), 23 pages. https://doi.org/10.1145/3591274

[43] Peipei Wang, Chris Brown, Jamie A. Jennings, and Kathryn T. Stolee. 2020. An Empirical Study on Regular Expression Bugs. In *Proceedings of the 17th International Conference on Mining Software Repositories* (Seoul, Republic of Korea) *(MSR '20)*. Association for Computing Machinery, New York, NY, USA, 103–113. https://doi.org/10.1145/3379597.3387464

[44] Peipei Wang, Chris Brown, Jamie A. Jennings, and Kathryn T. Stolee. 2022. Demystifying regular expression bugs: A comprehensive study on regular expression bug causes, fixes, and testing. *Empirical Softw. Engg.* 27, 1 (jan 2022), 35 pages. https://doi.org/10.1007/s10664-021-10033-1

[45] Tianyi Zhang, London Lowmanstone, Xinyu Wang, and Elena L. Glassman. 2020. Interactive Program Synthesis by Augmented Examples. In *Proceedings of the 33rd Annual ACM Symposium on User Interface Software and Technology* (Virtual Event, USA) *(UIST '20)*. Association for Computing Machinery, New York, NY, USA, 627–648. https://doi.org/10.1145/3379337.3415900

Received 20 February 2007; revised 12 March 2009; accepted 5 June 2009