What Is Decidable about String Constraints with the ReplaceAll Function

TAOLUE CHEN, Birkbeck, University of London, United Kingdom
YAN CHEN, State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences, China and University of Chinese Academy of Sciences, China
MATTHEW HAGUE, Royal Holloway, University of London, United Kingdom
ANTHONY W. LIN, University of Oxford, United Kingdom
ZHILIN WU, State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences, China

The theory of strings with concatenation has been widely argued as the basis of constraint solving for verifying string-manipulating programs. However, this theory is far from adequate for expressing many string constraints that are also needed in practice; for example, the use of regular constraints (pattern matching against a regular expression), and the string-replace function (replacing either the first occurrence or all occurrences of a “pattern” string constant/variable/regular expression by a “replacement” string constant/variable), among many others. Both regular constraints and the string-replace function are crucial for such applications as analysis of JavaScript (or more generally HTML5 applications) against cross-site scripting (XSS) vulnerabilities, which motivates us to consider a richer class of string constraints. The importance of the string-replace function (especially the replace-all facility) is increasingly recognised, which can be witnessed by the incorporation of the function in the input languages of several string constraint solvers.

Recently, it was shown that any theory of strings containing the string-replace function (even the most restricted version where pattern/replacement strings are both constant strings) becomes undecidable if we do not impose some kind of straight-line (aka acyclicity) restriction on the formulas. Despite this, the straight-line restriction is still practically sensible since this condition is typically met by string constraints that are generated by symbolic execution. In this paper, we provide the first systematic study of straight-line string constraints with the string-replace function and the regular constraints as the basic operations. We show that a large class of such constraints (i.e. when only a constant string or a regular expression is permitted in the pattern) is decidable. We note that the string-replace function, even under this restriction, is sufficiently powerful for expressing the concatenation operator and much more (e.g. extensions of regular expressions with string variables). This gives us the most expressive decidable logic containing concatenation, replace, and regular constraints under the same umbrella. Our decision procedure for the straight-line fragment follows an automata-theoretic approach, and is modular in the sense that the string-replace terms are removed one by one to generate more and more regular constraints, which can then be discharged by the state-of-the-art string constraint solvers. We also show that this fragment is, in a way, a maximal decidable subclass of the

Authors’ addresses: Taolue Chen, Department of Computer Science and Information Systems, Birkbeck, University of London, Malet Street, London, WC1E 7HX, United Kingdom, taolue@dcs.bbk.ac.uk; Yan Chen, State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences, China , University of Chinese Academy of Sciences, China; Matthew Hague, Department of Computer Science, Royal Holloway, University of London, Egham Hill, Egham, Surrey, TW20 0EX, United Kingdom, matthew.hague@rhul.ac.uk; Anthony W. Lin, Department of Computer Science, University of Oxford, Wolfson Buildin, Parks Road, Oxford, OX1 3QD, United Kingdom, anthony.lin@cs.ox.ac.uk; Zhilin Wu, State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences, China.

2475-1421/2018/1-ART3
https://doi.org/10.1145/3158091

straight-line fragment with string-replace and regular constraints. To this end, we show undecidability results for the following two extensions: (1) variables are permitted in the pattern parameter of the replace function, (2) length constraints are permitted.

CCS Concepts: • **Theory of computation → Automated reasoning**; **Verification by model checking**; **Program verification**; **Program analysis**; *Logic and verification*; Complexity classes;

Additional Key Words and Phrases: String Constraints, ReplaceAll, Decision Procedures, Constraint Solving, Straight-Line Programs

ACM Reference Format:
Taolue Chen, Yan Chen, Matthew Hague, Anthony W. Lin, and Zhilin Wu. 2018. What Is Decidable about String Constraints with the ReplaceAll Function . *Proc. ACM Program. Lang.* 2, POPL, Article 3 (January 2018), 29 pages. https://doi.org/10.1145/3158091

1 INTRODUCTION

The problem of automatically solving string constraints (aka satisfiability of logical theories over strings) has recently witnessed renewed interests [Abdulla et al. 2017, 2014; Bjørner et al. 2009; D'Antoni and Veanes 2013; Hooimeijer et al. 2011; Kiezun et al. 2012; Liang et al. 2014; Lin and Barceló 2016; Saxena et al. 2010; Trinh et al. 2014, 2016; Veanes et al. 2012; Wang et al. 2016; Yu et al. 2014; Zheng et al. 2013] because of important applications in the analysis of string-manipulating programs. For example, program analysis techniques like symbolic execution [Cadar et al. 2006; Godefroid et al. 2005; King 1976; Sen et al. 2013] would systematically explore executions in a program and collect symbolic path constraints, which could then be solved using a constraint solver and used to determine which location in the program to continue exploring. To successfully apply a constraint solver in this instance, it is crucial that the constraint language precisely models the data types in the program, along with the data-type operations used. In the context of string-manipulating programs, this could include concatenation, regular constraints (i.e. pattern matching against a regular expression), string-length functions, and the string-replace functions, among many others.

Perhaps the most well-known theory of strings for such applications as the analysis of string-manipulating programs is the theory of strings with concatenation (aka *word equations*), whose decidability was shown by Makanin [Makanin 1977] in 1977 after it was open for many years. More importantly, this theory remains decidable even when regular constraints are incorporated into the language [Schulz 1990]. However, whether adding the string-length function preserves the decidability remains a long-standing open problem [Büchi and Senger 1990; Ganesh et al. 2012].

Another important string operation—especially in popular scripting languages like Python, JavaScript, and PHP—is the *string-replace function*, which may be used to replace either the *first* occurrence or *all* occurrences of a string (a string constant/variable, or a regular expression) by another string (a string constant/variable). The replace function (especially the replace-all functionality) is omnipresent in HTML5 applications [Lin and Barceló 2016; Trinh et al. 2016; Yu et al. 2014]. For example, a standard industry defense against cross-site scripting (XSS) vulnerabilities includes sanitising untrusted strings before adding them into the DOM (Document Object Model) or the HTML document. This is typically done by various metacharacter-escaping mechanisms (see, for instance, [Hooimeijer et al. 2011; Kern 2014; Williams et al. 2017]). An example of such a mechanism is backslash-escape, which replaces *every occurrence* of quotes and double-quotes (i.e. `'` and `"`) in the string by `\'` and `\"`. In addition to sanitisers, common JavaScript functionalities like `document.write()` and `innerHTML` apply an *implicit browser transduction* — which decodes HTML codes (e.g. `&#39;` is replaced by `'`) in the input string — before inserting the input string into the DOM. Both of these examples can be expressed by (perhaps multiple) applications of the string-replace function. Moreover, although these examples replace constants by constants, the popularity of template systems such as Mustache [Wanstrath 2009] and Closure Templates [Google 2015] demonstrate the need for replacements involving variables. Using Mustache, a web-developer, for example, may define an HTML fragment with placeholders that is instantiated with user data during the construction of the delivered page.

Example 1.1. We give a simple example demonstrating a (naive) XSS vulnerability to illustrate the use of string-replace functions. Consider the HTML fragment below.

```
<h1> User <span onMouseOver="popupText('{{bio}}')">{{userName}}</span> </h1>
```

This HTML fragment is a template as might be used with systems such as Mustache to display a user on a webpage. For each user that is to be displayed – with their username and biography stored in variables *user* and *bio* respectively – the string `{{userName}}` will be replaced by *user* and the string `{{bio}}` will be replaced by *bio*. For example, a user `Amelia` with biography `Amelia was born in 1979...` would result in the HTML below.

```
<h1> User
     <span onMouseOver="popupText('Amelia was born in 1979...')">
         Amelia </span> </h1>
```

This HTML would display `User Amelia`, and, when the mouse is placed over `Amelia`, her biography would appear, thanks to the `onMouseOver` attribute in the `span` element.

Unfortunately, this template could be insecure if the user biography is not adequately sanitised: A user could enter a malicious biography, such as `'); alert('Boo!'); alert('` which would cause the following instantiation of the `span` element[1].

```
<span onMouseOver="popupText(''); alert('Boo!'); alert('')">
```

Now, when the mouse is placed over the user name, the malicious JavaScript `alert('Boo!')` is executed.

The presence of such malicious injections of code can be detected using string constraint solving and XSS *attack patterns* given as regular expressions [Balzarotti et al. 2008; Saxena et al. 2010; Yu et al. 2014]. For our example, given an attack pattern P and template *temp*, we would generate the constraint

$$x_1 = \mathsf{replaceAll}(temp, \texttt{\{\{userName\}\}}, user) \wedge x_2 = \mathsf{replaceAll}(x_1, \texttt{\{\{bio\}\}}, bio) \wedge x_2 \in P$$

which would detect if the HTML generated by instantiating the template is susceptible to the attack identified by P. □

In general, the string-replace function has three parameters, and in the current mainstream language such as Python and JavaScript, *all of the three parameters can be inserted as string variables.* As result, when we perform program analysis for, for instance, detecting security vulnerabilities as described above, one often obtains string constraints of the form $z = \mathsf{replaceAll}(x, p, y)$, where x, y are string constants/variables, and p is either a string constant/variable or a regular expression. Such a constraint means that z is obtained by replacing all occurrences of p in x with y. For convenience, we call x, p, y as the *subject*, the *pattern*, and the *replacement* parameters respectively.

The replaceAll function is a powerful string operation that goes beyond the expressiveness of concatenation. (On the contrary, as we will see later, concatenation can be expressed by the replaceAll function easily.) It was shown in a recent POPL paper [Lin and Barceló 2016] that any theory of strings containing the string-replace function (even the most restricted version

[1]Readers familiar with Mustache and Closure Templates may expect single quotes to be automatically escaped. However, we have tested our example with the latest versions of mustache.js [Lehnardt and contributors 2015] and Closure Templates [Google 2015] (as of July 2017) and observed that the exploit is not disarmed by their automatic escaping features.

where pattern/replacement strings are both constant strings) becomes undecidable if we do not impose some kind of *straight-line restriction*[2] on the formulas. Nonetheless, as already noted in [Lin and Barceló 2016], the straight-line restriction is reasonable since it is typically satisfied by constraints that are generated by symbolic execution, e.g., all constraints in the standard Kaluza benchmarks [Saxena et al. 2010] with 50,000+ test cases generated by symbolic execution on JavaScript applications were noted in [Ganesh et al. 2012] to satisfy this condition. Intuitively, as elegantly described in [Bjørner et al. 2009], constraints from symbolic execution on string-manipulating programs can be viewed as the problem of path feasibility over loopless string-manipulating programs S with variable assignments and assertions, i.e., generated by the grammar

$$S ::= y := f(x_1, \ldots, x_n) \mid \mathbf{assert}(g(x_1, \ldots, x_n)) \mid S_1; S_2$$

where $f : (\Sigma^*)^n \to \Sigma^*$ and $g : (\Sigma^*)^n \to \{0, 1\}$ are some string functions. Straight-line programs with assertions can be obtained by turning such programs into a Static Single Assignment (SSA) form (i.e. introduce a new variable on the left hand side of each assignment). A partial decidability result can be deduced from [Lin and Barceló 2016] for the straight-line fragment of the theory of strings, where (1) f in the above grammar is either a concatenation of string constants and variables, or the replaceAll function where *the pattern and the replacement are both string constants*, and (2) g is a boolean combination of regular constraints. In fact, the decision procedure therein admits finite-state transducers, which subsume only the aforementioned simple form of the replaceAll function. The decidability boundary of the straight-line fragment involving the replaceAll function in its general form (e.g., when the replacement parameter is a variable) remains open.

Contribution. We investigate the decidability boundary of the theory SL[replaceAll] of strings involving the replaceAll function and regular constraints, with the straight-line restriction introduced in [Lin and Barceló 2016]. We provide a decidability result for a large fragment of SL[replaceAll], which is sufficiently powerful to express the concatenation operator. We show that this decidability result is in a sense maximal by showing that several important natural extensions of the logic result in undecidability. We detail these results below:

- If the pattern parameters of the replaceAll function are allowed to be variables, then the satisfiability of SL[replaceAll] is undecidable (cf. Proposition 4.1).
- If the pattern parameters of the replaceAll function are regular expressions, then the satisfiability of SL[replaceAll] is decidable and in EXPSPACE (cf. Theorem 4.2). In addition, we show that the satisfiability problem is PSPACE-complete for several cases that are meaningful in practice (cf. Corollary 4.7). This strictly generalises the decidability result in [Lin and Barceló 2016] of the straight-line fragment with concatenation, regular constraints, and the replaceAll function where patterns/replacement parameters are constant strings.
- If SL[replaceAll], where the pattern parameter of the replaceAll function is a constant letter, is extended with the string-length constraint, then satisfiability becomes undecidable again. In fact, this undecidability can be obtained with either integer constraints, character constraints, or constraints involving the IndexOf function (cf. Theorem 9.4 and Proposition 9.6).

Our decision procedure for SL[replaceAll] where the pattern parameters of the replaceAll function are regular expressions follows an automata-theoretic approach. The key idea can be illustrated as follows. Let us consider the simple formula $C \equiv x = \mathsf{replaceAll}(y, a, z) \wedge x \in e_1 \wedge y \in e_2 \wedge z \in e_3$. Suppose that $\mathcal{A}_1, \mathcal{A}_2, \mathcal{A}_3$ are the nondeterministic finite state automata corresponding to e_1, e_2, e_3 respectively. We effectively eliminate the use of replaceAll by nondeterministically generating from $\mathcal{A}_1$ a new regular constraint $\mathcal{A}'_2$ for y as well as a new regular constraint $\mathcal{A}'_3$ for z. These

[2]Similar notions that appear in the literature of string constraints (without replace) include acyclicity [Abdulla et al. 2014] and solved form [Ganesh et al. 2012]

constraints incorporate the effect of the replaceAll function (i.e. all regular constraints are on the "source" variables). Then, the satisfiability of C is turned into testing the nonemptiness of the intersection of $\mathcal{A}_2$ and $\mathcal{A}'_2$, as well as the nonemptiness of the intersection of $\mathcal{A}_3$ and $\mathcal{A}'_3$. When there are multiple occurrences of the replaceAll function, this process can be iterated. Our decision procedure enjoys the following advantages:

- It is automata-theoretic and built on clean automaton constructions, moreover, when the formula is satisfiable, a solution can be synthesised. For example, in the aforementioned XSS vulnerability detection example, one can synthesise the values of the variables *user* and *bio* for a potential attack.
- The decision procedure is modular in that the replaceAll terms are removed one by one to generate more and more regular constraints (emptiness of the intersection of regular constraints could be efficiently handled by state-of-the-art solvers like [Wang et al. 2016]).
- The decision procedure requires exponential space (thus double exponential time), but under assumptions that are reasonable in practice, the decision procedure uses only polynomial space, which is not worse than other string logics (which can encode the PSPACE-complete problem of checking emptiness of the intersection of regular constraints).

Organisation. This paper is organised as follows: Preliminaries are given in Section 2. The core string language is defined in Section 3. The main results of this paper are summarised in Section 4. The decision procedure is presented in Section 6-8, case by case. The extensions of the core string language are investigated in Section 9. The related work can be found in Section 10. The full version contains missing proofs and additional examples.

2 PRELIMINARIES

General Notation. Let $\mathbb{Z}$ and $\mathbb{N}$ denote the set of integers and natural numbers respectively. For $k \in \mathbb{N}$, let $[k] = \{1, \cdots, k\}$. For a vector $\vec{x} = (x_1, \cdots, x_n)$, let $|\vec{x}|$ denote the length of $\vec{x}$ (i.e., n) and $\vec{x}[i]$ denote x_i for each $i \in [n]$.

Regular Languages. Fix a finite *alphabet* Σ. Elements in Σ^* are called *strings*. Let ε denote the empty string and $\Sigma^+ = \Sigma^* \setminus \{\varepsilon\}$. We will use $a, b, \cdots$ to denote letters from Σ and $u, v, w, \cdots$ to denote strings from Σ^*. For a string $u \in \Sigma^*$, let $|u|$ denote the *length* of u (in particular, $|\varepsilon| = 0$). A *position* of a nonempty string u of length n is a number $i \in [n]$ (Note that the first position is 1, instead of 0). In addition, for $i \in [|u|]$, let $u[i]$ denote the i-th letter of u. For two strings u_1, u_2, we use $u_1 \cdot u_2$ to denote the *concatenation* of u_1 and u_2, that is, the string v such that $|v| = |u_1| + |u_2|$ and for each $i \in [|u_1|]$, $v[i] = u_1[i]$ and for each $i \in |u_2|$, $v[|u_1| + i] = u_2[i]$. Let u, v be two strings. If $v = u \cdot v'$ for some string v', then u is said to be a *prefix* of v. In addition, if $u \neq v$, then u is said to be a *strict* prefix of v. If u is a prefix of v, that is, $v = u \cdot v'$ for some string v', then we use $u^{-1}v$ to denote v'. In particular, $\varepsilon^{-1}v = v$.

A *language* over Σ is a subset of Σ^*. We will use $L_1, L_2, \ldots$ to denote languages. For two languages L_1, L_2, we use $L_1 \cup L_2$ to denote the union of L_1 and L_2, and $L_1 \cdot L_2$ to denote the concatenation of L_1 and L_2, that is, the language $\{u_1 \cdot u_2 \mid u_1 \in L_1, u_2 \in L_2\}$. For a language L and $n \in \mathbb{N}$, we define L^n, the *iteration* of L for n times, inductively as follows: $L^0 = \{\varepsilon\}$ and $L^n = L \cdot L^{n-1}$ for $n > 0$. We also use L^* to denote the iteration of L for arbitrarily many times, that is, $L^* = \bigcup\limits_{n \in \mathbb{N}} L^n$. Moreover, let $L^+ = \bigcup\limits_{n \in \mathbb{N} \setminus \{0\}} L^n$.

Definition 2.1 (Regular expressions RegExp*).*

$$e \overset{\text{def}}{=} \emptyset \mid \varepsilon \mid a \mid e + e \mid e \circ e \mid e^*, \text{ where } a \in \Sigma.$$

Since $+$ is associative and commutative, we also write $(e_1 + e_2) + e_3$ as $e_1 + e_2 + e_3$ for brevity. We use the abbreviation $e^+ \equiv e \circ e^*$. Moreover, for $\Gamma = \{a_1, \cdots, a_n\} \subseteq \Sigma$, we use the abbreviations $\Gamma \equiv a_1 + \cdots + a_n$ and $\Gamma^* \equiv (a_1 + \cdots + a_n)^*$.

We define $\mathcal{L}(e)$ to be the language defined by e, that is, the set of strings that match e, inductively as follows: $\mathcal{L}(\emptyset) = \emptyset$, $\mathcal{L}(\varepsilon) = \{\varepsilon\}$, $\mathcal{L}(a) = \{a\}$, $\mathcal{L}(e_1 + e_2) = \mathcal{L}(e_1) \cup \mathcal{L}(e_2)$, $\mathcal{L}(e_1 \circ e_2) = \mathcal{L}(e_1) \cdot \mathcal{L}(e_2)$, $\mathcal{L}(e_1^*) = (\mathcal{L}(e_1))^*$. In addition, we use $|e|$ to denote the number of symbols occurring in e.

A *nondeterministic finite automaton* (NFA) $\mathcal{A}$ on Σ is a tuple (Q, δ, q_0, F), where Q is a finite set of *states*, $q_0 \in Q$ is the *initial* state, $F \subseteq Q$ is the set of *final* states, and $\delta \subseteq Q \times \Sigma \times Q$ is the *transition relation*. For a string $w = a_1 \dots a_n$, a *run* of $\mathcal{A}$ on w is a state sequence $q_0 \dots q_n$ such that for each $i \in [n]$, $(q_{i-1}, a_i, q_i) \in \delta$. A run $q_0 \dots q_n$ is *accepting* if $q_n \in F$. A string w is *accepted* by $\mathcal{A}$ if there is an accepting run of $\mathcal{A}$ on w. We use $\mathcal{L}(\mathcal{A})$ to denote the language defined by $\mathcal{A}$, that is, the set of strings accepted by $\mathcal{A}$. We will use $\mathcal{A}, \mathcal{B}, \cdots$ to denote NFAs. For a string $w = a_1 \dots a_n$, we also use the notation $q_1 \xrightarrow[\mathcal{A}]{w} q_{n+1}$ to denote the fact that there are $q_2, \dots, q_n \in Q$ such that for each $i \in [n]$, $(q_i, a_i, q_{i+1}) \in \delta$. For an NFA $\mathcal{A} = (Q, \delta, q_0, F)$ and $q, q' \in Q$, we use $\mathcal{A}(q, q')$ to denote the NFA obtained from $\mathcal{A}$ by changing the initial state to q and the set of final states to $\{q'\}$. The *size* of an NFA $\mathcal{A} = (Q, \delta, q_0, F)$, denoted by $|\mathcal{A}|$, is defined as $|Q|$, the number of states. For convenience, we will also call an NFA without initial and final states, that is, a pair (Q, δ), as a *transition graph*.

It is well-known (e.g. see [Hopcroft and Ullman 1979]) that regular expressions and NFAs are expressively equivalent, and generate precisely all *regular languages*. In particular, from a regular expression, an equivalent NFA can be constructed in linear time. Moreover, regular languages are closed under Boolean operations, i.e., union, intersection, and complementation. In particular, given two NFA $\mathcal{A}_1 = (Q_1, \delta_1, q_{0,1}, F_1)$ and $\mathcal{A}_2 = (Q_2, \delta_2, q_{0,2}, F_2)$ on Σ, the intersection $\mathcal{L}(\mathcal{A}_1) \cap \mathcal{L}(\mathcal{A}_2)$ is recognised by the *product automaton* $\mathcal{A}_1 \times \mathcal{A}_2$ of $\mathcal{A}_1$ and $\mathcal{A}_2$ defined as $(Q_1 \times Q_2, \delta, (q_{0,1}, q_{0,2}), F_1 \times F_2)$, where δ comprises the transitions $((q_1, q_2), a, (q_1', q_2'))$ such that $(q_1, a, q_1') \in \delta_1$ and $(q_2, a, q_2') \in \delta_2$.

Graph-Theoretical Notation. A DAG (*directed acyclic graph*) G is a finite directed graph (V, E) with no directed cycles, where V (resp. $E \subseteq V \times V$) is a set of vertices (resp. edges). Equivalently, a DAG is a directed graph that has a topological ordering, which is a sequence of the vertices such that every edge is directed from an earlier vertex to a later vertex in the sequence. An edge (v, v') in G is called an *incoming* edge of v' and an *outgoing* edge of v. If $(v, v') \in E$, then v' is called a *successor* of v and v is called a *predecessor* of v'. A *path* π in G is a sequence $v_0 e_1 v_1 \cdots v_{n-1} e_n v_n$ such that for each $i \in [n]$, we have $e_i = (v_{i-1}, v_i) \in E$. The *length* of the path π is the number n of edges in π. If there is a path from v to v' (resp. from v' to v) in G, then v' is said to be *reachable* (resp. *co-reachable*) from v in G. If v is reachable from v' in G, then v' is also called an *ancestor* of v in G. In addition, an edge (v', v'') is said to be reachable (resp. co-reachable) from v if v' is reachable from v (resp. v'' is co-reachable from v). The *in-degree* (resp. *out-degree*) of a vertex v is the number of incoming (resp. outgoing) edges of v. A *subgraph* G' of $G = (V, E)$ is a directed graph (V', E') with $V' \subseteq V$ and $E' \subseteq E$. Let G' be a subgraph of G. Then $G \setminus G'$ is the graph obtained from G by removing all the edges in G'.

Computational Complexity. In this paper, we study not only decidability but also the complexity of string logics. In particular, we shall deal with the following computational complexity classes (see [Hopcroft and Ullman 1979] for more details): PSPACE (problems solvable in polynomial space and thus in exponential time), and EXPSPACE (problems solvable in exponential space and thus in double exponential time). Verification problems that have complexity PSPACE or beyond (see [Baier and Katoen 2008] for a few examples) have substantially benefited from techniques such as symbolic model checking [McMillan 1993].

3 THE CORE CONSTRAINT LANGUAGE

In this section, we define a general string constraint language that supports concatenation, the replaceAll function, and regular constraints. Throughout this section, we fix an alphabet Σ.

3.1 Semantics of the replaceAll Function

To define the semantics of the replaceAll function, we note that the function encompasses three parameters: the first parameter is the *subject* string, the second parameter is a *pattern* that is a string or a regular expression, and the third parameter is the *replacement* string. When the pattern parameter is a string, the semantics is somehow self-explanatory. However, when it is a regular expression, there is no consensus on the semantics even for the mainstream programming languages such as Python and Javascript. This is particularly the case when interpreting the union (aka alternation) operator in regular expressions or performing a replaceAll with a pattern that matches ε. In this paper, we mainly focus on the semantics of *leftmost and longest matching*. Our handling of ε matches is consistent with our testing of the implementation in Python and the `sed` command with the `--posix` flag. We also assume union is commutative (e.g. $\mathsf{replaceAll}(aa, a + aa, b) = \mathsf{replaceAll}(aa, aa + a, b) = b$) as specified by POSIX, but often ignored in practice (where bb is a common result in the former case).

Definition 3.1. Let u, v be two strings such that $v = v_1 u v_2$ for some v_1, v_2 and e be a regular expression. We say that u is the *leftmost and longest* matching of e in v if one of the following two conditions hold,

- case $\varepsilon \notin \mathcal{L}(e)$:
  (1) leftmost: $u \in \mathcal{L}(e)$, and $(v_1')^{-1}v \notin \mathcal{L}(e \circ \Sigma^*)$ for every strict prefix v_1' of v_1,
  (2) longest: for every nonempty prefix v_2' of v_2, $u \cdot v_2' \notin \mathcal{L}(e)$.
- case $\varepsilon \in \mathcal{L}(e)$:
  (1) leftmost: $u \in \mathcal{L}(e)$, and $v_1 = \varepsilon$,
  (2) longest: for every nonempty prefix v_2' of v_2, $u \cdot v_2' \notin \mathcal{L}(e)$.

Example 3.2. Let us first consider $\Sigma = \{0, 1\}$, $v = 1010101$, $v_1 = 1$, $u = 010$, $v_2 = 101$, and $e = 0^*01(0^* + 1^*)$. Then $v = v_1 u v_2$, and the leftmost and longest matching of e in v is u. This is because $u \in \mathcal{L}(e)$, $\varepsilon^{-1}v = v \notin \mathcal{L}(e \circ \Sigma^*)$ (notice that v_1 has only one strict prefix, i.e. ε), and none of $u1 = 0101$, $u10 = 01010$, and $u101 = 010101$ belong to $\mathcal{L}(e)$ (notice that v_2 has three nonempty prefixes, i.e. $1, 10, 101$). For another example, let us consider $\Sigma = \{a, b, c\}$, $v = baac$, $v_1 = \varepsilon$, $u = \varepsilon$, $v_2 = v$, and $e = a^*$. Then $v = v_1 u v_2$ and the leftmost and longest matching of e in v is u. This is because $u \in \mathcal{L}(e)$, $v_1 = \varepsilon$, and $b, ba, baa, baac \notin \mathcal{L}(e)$. On the other hand, similarly, one can verify that the leftmost and longest matching of $e = a^*$ in $v = aac$ is $u = aa$.

Definition 3.3. The semantics of $\mathsf{replaceAll}(u, e, v)$, where u, v are strings and e is a regular expression, is defined inductively as follows:

- if $u \notin \mathcal{L}(\Sigma^* \circ e \circ \Sigma^*)$, that is, u does *not* contain any substring from $\mathcal{L}(e)$, then $\mathsf{replaceAll}(u, e, v) = u$,
- otherwise,
  - if $\varepsilon \in \mathcal{L}(e)$ and u is the leftmost and longest matching of e in u, then $\mathsf{replaceAll}(u, e, v) = v$,
  - if $\varepsilon \in \mathcal{L}(e)$, $u = u_1 \cdot a \cdot u_2$, u_1 is the leftmost and longest matching of e in u, and $a \in \Sigma$, then $\mathsf{replaceAll}(u, e, v) = v \cdot a \cdot \mathsf{replaceAll}(u_2, e, v)$,
  - if $\varepsilon \notin \mathcal{L}(e)$, $u = u_1 \cdot u_2 \cdot u_3$, and u_2 is the leftmost and longest matching of e in u, then $\mathsf{replaceAll}(u, e, v) = u_1 \cdot v \cdot \mathsf{replaceAll}(u_3, e, v)$.

Example 3.4. At first, $\mathsf{replaceAll}(abab, ab, d) = d \cdot \mathsf{replaceAll}(ab, ab, d) = dd \cdot \mathsf{replaceAll}(\epsilon, ab, d) = dd \cdot \varepsilon = dd$ and $\mathsf{replaceAll}(baac, a^+, b) = bbc$. In addition, $\mathsf{replaceAll}(aaaa, \text{“”}, d) = dadadadad$ and

$\mathsf{replaceAll}(baac, a^*, b) = bbbcb$. The argument for $\mathsf{replaceAll}(baac, a^*, b) = bbbcb$ proceeds as follows: The leftmost and longest matching of a^* in $baac$ is $u_1 = \varepsilon$, where $baac = u_1 \cdot b \cdot u_2$ and $u_2 = aac$. Then $\mathsf{replaceAll}(baac, a^*, b) = b \cdot b \cdot \mathsf{replaceAll}(aac, a^*, b)$. Since aa is the leftmost and longest matching of a^* in aac, we have $\mathsf{replaceAll}(aac, a^*, b) = b \cdot c \cdot \mathsf{replaceAll}(\varepsilon, a^*, b) = bcb$. Therefore, we get $\mathsf{replaceAll}(baac, a^*, b) = bbbcb$. (The readers are invited to test this in Python and sed.)

3.2 Straight-Line String Constraints With the replaceAll Function

We consider the String data type Str, and assume a countable set of variables $x, y, z, \cdots$ of Str.

Definition 3.5 (Relational and regular constraints). Relational constraints and regular constraints are defined by the following rules,

$$\begin{array}{rcll} s & \overset{\text{def}}{=} & x \mid u & \text{(string terms)} \\ p & \overset{\text{def}}{=} & x \mid e & \text{(pattern terms)} \\ \varphi & \overset{\text{def}}{=} & x = s \circ s \mid x = \mathsf{replaceAll}(s, p, s) \mid \varphi \wedge \varphi & \text{(relational constraints)} \\ \psi & \overset{\text{def}}{=} & x \in e \mid \psi \wedge \psi & \text{(regular constraints)} \end{array}$$

where x is a string variable, $u \in \Sigma^*$ and e is a regular expression over Σ.

For a formula φ (resp. ψ), let $\mathsf{Vars}(\varphi)$ (resp. $\mathsf{Vars}(\psi)$) denote the set of variables occurring in φ (resp. ψ). Given a relational constraint φ, a variable x is called a *source variable* of φ if φ *does not* contain a conjunct of the form $x = s_1 \circ s_2$ or $x = \mathsf{replaceAll}(-, -, -)$.

We then notice that, with the replaceAll function in its general form, the concatenation operation is in fact redundant.

Proposition 3.6. *The concatenation operation ($\circ$) can be simulated by the* replaceAll *function.*

Proof. It is sufficient to observe that a relational constraint $x = s_1 \circ s_2$ can be rewritten as

$$x' = \mathsf{replaceAll}(ab, a, s_1) \wedge x = \mathsf{replaceAll}(x', b, s_2),$$

where a, b are two fresh letters. □

In light of Proposition 3.6, in the sequel, we will *dispense the concatenation operator* mostly and focus on **the string constraints that involve the** replaceAll **function only**.

Another example to show the power of the replaceAll function is that it can simulate the extension of regular expressions with string variables, which is supported by the mainstream scripting languages like Python, Javascript, and PHP. For instance, $x \in y^*$ can be expressed by $x = \mathsf{replaceAll}(x', a, y) \wedge x' \in a^*$, where x' is a fresh variable and a is a fresh letter.

The generality of the constraint language makes it undecidable, even in very simple cases. To retain decidability, we follow [Lin and Barceló 2016] and focus on the "straight-line fragment" of the language. This straight-line fragment captures the structure of straight-line string-manipulating programs with the replaceAll string operation.

Definition 3.7 (Straight-line relational constraints). A relational constraint φ with the replaceAll function is straight-line, if $\varphi \overset{\text{def}}{=} \bigwedge_{1 \leq i \leq m} x_i = P_i$ such that

- $x_1, \ldots, x_m$ are mutually distinct,
- for each $i \in [m]$, all the variables in P_i are either source variables, or variables from $\{x_1, \ldots, x_{i-1}\}$,

Remark 3.8. *Checking whether a relational constraint φ is straight-line can be done in linear time.*

Definition 3.9 (Straight-line string constraints). A straight-line string constraint C with the replaceAll function (denoted by SL[replaceAll]) is defined as $\varphi \wedge \psi$, where

- φ is a straight-line relational constraint with the replaceAll function, and
- ψ is a regular constraint.

Example 3.10. The following string constraint belongs to SL[replaceAll]:

$$C \equiv x_2 = \mathsf{replaceAll}(x_1, 0, y_1) \wedge x_3 = \mathsf{replaceAll}(x_2, 1, y_2) \wedge x_1 \in \{0,1\}^* \wedge y_1 \in 1^* \wedge y_2 \in 0^*.$$

4 THE SATISFIABILITY PROBLEM

In this paper, we focus on the satisfiability problem of SL[replaceAll], which is formalised as follows.

$$\boxed{\text{Given an SL[replaceAll] constraint } C\text{, decide whether } C \text{ is satisfiable.}}$$

To approach this problem, we identify several fragments of SL[replaceAll], depending on whether the pattern and the replacement parameters are constants or variables. We shall investigate extensively the satisfiability problem of the fragments of SL[replaceAll].

We begin with the case where the pattern parameters of the replaceAll terms are variables. It turns out that in this case the satisfiability problem of SL[replaceAll] is undecidable. The proof is by a reduction from Post's Correspondence Problem. Due to space constraints we relegate the proof to the full version.

Proposition 4.1. *The satisfiability problem of* SL[replaceAll] *is undecidable, if the pattern parameters of the* replaceAll *terms are allowed to be variables.*

In light of Proposition 4.1, we shall focus on the case that the pattern parameters of the replaceAll terms are constants, being a single letter, a constant string, or a regular expression. The main result of the paper is summarised as the following Theorem 4.2.

Theorem 4.2. *The satisfiability problem of* SL[replaceAll] *is decidable in EXPSPACE, if the pattern parameters of the* replaceAll *terms are regular expressions.*

The following three sections are devoted to the proof of Theorem 4.2.

- We start with the *single-letter* case that the pattern parameters of the replaceAll terms are single letters (Section 6),
- then consider the *constant-string* case that the pattern parameters of the replaceAll terms are constant strings (Section 7),
- and finally the *regular-expression* case that the pattern parameters of the replaceAll terms are regular expressions (Section 8).

We first introduce a graphical representation of SL[replaceAll] formulae as follows.

Definition 4.3 (Dependency graph). Suppose $C = \varphi \wedge \psi$ is an SL[replaceAll] formula where the pattern parameters of the replaceAll terms are regular expressions. Define the *dependency graph* of C as $G_C = (\mathsf{Vars}(\varphi), E_C)$, such that for each $i \in [m]$, if $x_i = \mathsf{replaceAll}(z, e_i, z')$, then $(x_i, (\mathsf{l}, e_i), z) \in E_C$ and $(x_i, (\mathsf{r}, e_i), z') \in E_C$. A final (resp. initial) vertex in G_C is a vertex in G_C without successors (resp. predecessors). The edges labelled by (l, e_i) and (r, e_i) are called the l-edges and r-edges respectively. The *depth* of G_C is the maximum length of the paths in G_C. In particular, if φ is empty, then the depth of G_C is zero.

Note that G_C is a DAG where the out-degree of each vertex is two or zero.

Definition 4.4 (Diamond index and l*-length).* Let C be an SL[replaceAll] formula and $G_C = (\mathsf{Vars}(\varphi), E_C)$ be its dependency graph. A *diamond* Δ in G_C is a pair of vertex-disjoint simple paths from z to z' for some $z, z' \in \mathsf{Vars}(\varphi)$. The vertices z and z' are called the *source* and *destination* vertex of the diamond respectively. A diamond Δ_2 with the source vertex z_2 and destination vertex z'_2 is said to be reachable from another diamond Δ_1 with the source vertex z_1 and destination vertex z'_1 if z_2 is reachable from z'_1 (possibly $z_2 = z'_1$). The *diamond index* of G_C, denoted by $\mathsf{Idx}_{\mathsf{dmd}}(G_C)$, is defined as the maximum length of the diamond sequences $\Delta_1 \cdots \Delta_n$ in G_C such that for each $i \in [n-1]$, Δ_{i+1} is reachable from Δ_i. The l-*length* of a path in G_C is the number of l-edges in the path. The l-length of G_C, denoted by $\mathsf{Len}_{\mathsf{lft}}(G_C)$, is the maximum l-length of paths in G_C.

For each dependency graph G_C, since each diamond uses at least one l-edge, we know that $\mathsf{Idx}_{\mathsf{dmd}}(G_C) \leq \mathsf{Len}_{\mathsf{lft}}(G_C)$.

Proposition 4.5. *Let C be an* SL[replaceAll] *formula and $G_C = (\mathsf{Vars}(\varphi), E_C)$ be its dependency graph. For each pair of distinct vertices z, z' in G_C, there are at most $(|\mathsf{Vars}(\varphi)||E_C|)^{O(\mathsf{Idx}_{\mathsf{dmd}}(G_C))}$ different paths from z to z'.*

It follows from Proposition 4.5 that for a class of SL[replaceAll] formulae C such that $\mathsf{Idx}_{\mathsf{dmd}}(G_C)$ is bounded by a constant c, there are polynomially many different paths between each pair of distinct vertices in G_C.

Example 4.6. Let G_C be the dependency graph illustrated in Figure 1. It is easy to see that $\mathsf{Idx}_{\mathsf{dmd}}(G_C)$ is 3. In addition, there are $2^3 = 8$ paths from x_1 to y_1. If we generalise G_C in Figure 1 to a dependency graph comprising n diamonds from x_1 to $x_2, \cdots$, from x_{n-1} to x_n, and from x_n to y_1 respectively, then the diamond index of the resulting dependency graph is n and there are 2^n paths from x_1 to y_1 in the graph.

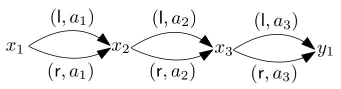

$$\begin{tikzcd} x_1 \arrow[r, bend left, "(\mathsf{l}, a_1)"] \arrow[r, bend right, "(\mathsf{r}, a_1)"'] & x_2 \arrow[r, bend left, "(\mathsf{l}, a_2)"] \arrow[r, bend right, "(\mathsf{r}, a_2)"'] & x_3 \arrow[r, bend left, "(\mathsf{l}, a_3)"] \arrow[r, bend right, "(\mathsf{r}, a_3)"'] & y_1 \end{tikzcd}$$

Fig. 1. The diamond index and the number of paths in G_C

In Section 6–8, we will apply a refined analysis of the complexity of the decision procedures for proving Theorem 4.2 and get the following results.

Corollary 4.7. *The satisfiability problem is PSPACE-complete for the following fragments of* SL[replaceAll]*:*

- *the single-letter case, plus the condition that the diamond indices of the dependency graphs are bounded by a constant c,*
- *the constant-string case, plus the condition that the l-lengths of the dependency graphs are bounded by a constant c,*
- *the regular-expression case, plus the condition that the l-lengths of the dependency graphs are at most* 1.

Corollary 4.7 partially justifies our choice to present the decision procedures for the single-letter, constant-string, and regular-expression case separately. Intuitively, when the pattern parameters of the replaceAll terms become less restrictive, the decision procedures become more involved, and more constraints should be imposed on the dependency graphs in order to achieve the PSPACE upper-bound. The PSPACE lower-bound follows from the observation that nonemptiness of the intersection of the regular expressions $e_1, \cdots, e_n$ over the alphabet $\{0, 1\}$, which is a PSPACE-complete problem, can be reduced to the satisfiability of the formula $x \in e_1 \wedge \cdots \wedge x \in e_n$, which falls into all fragments of SL[replaceAll] specified in Corollary 4.7. At last, we remark that the restrictions in Corollary 4.7 are partially inspired by the benchmarks in practice. Diamond indices (intuitively, the "nesting depth" of replaceAll(x, a, x)) are likely to be small in practice because the constraints like replaceAll(x, a, x) are rather artificial and rarely occur in practice. Moreover, the l-length reflects the nesting depth of replaceall w.r.t. the first parameter, which is also likely to be small. Finally, for string constraints with concatenation and replaceAll where pattern/replacement parameters are constants, the diamond index is no greater than the "dimension" defined in [Lin and Barceló 2016], where it was shown that existing benchmarks mostly have "dimensions" at most three for such string constraints.

5 OUTLINE OF DECISION PROCEDURES

We describe our decision procedure across three sections (Section 6–Section 8). This means the ideas can be introduced in a step-by-step fashion, which we hope helps the reader. In addition, by presenting separate algorithms, we can give the fine-grained complexity analysis required to show Corollary 4.7. We first outline the main ideas needed by our approach.

We will use automata-theoretic techniques. That is, we make use of the fact that regular expressions can be represented as NFAs. We can then consider a very simple string expression, which is a single regular constraint $x \in e$. It is well-known that an NFA $\mathcal{A}$ can be constructed that is equivalent to e. We can also test in LOGSPACE whether there is some word w accepted by $\mathcal{A}$. If this is the case, then this word can be assigned to x, giving a satisfying assignment to the constraint. If this is not the case, then there is no satisfying assignment.

A more complex case is a conjunction of several constraints of the form $x \in e$. If the constraints apply to different variables, they can be treated independently to find satisfying assignments. If the constraints apply to the same variable, then they can be merged into a single NFA. Intuitively, take $x \in e_1 \wedge x \in e_2$ and $\mathcal{A}_1$ and $\mathcal{A}_2$ equivalent to e_1 and e_2 respectively. We can use the fact that NFA are closed under intersection a check if there is a word accepted by $\mathcal{A}_1 \times \mathcal{A}_2$. If this is the case, we can construct a satisfying assignment to x from an accepting run of $\mathcal{A}_1 \times \mathcal{A}_2$.

In the general case, however, variables are not independent, but may be related by a use of replaceAll. In this case, we perform a kind of replaceAll *elimination*. That is, we successively remove instances of replaceAll from the constraint, building up an expanded set of regular constraints (represented as automata). Once there are no more instances of replaceAll we can solve the regular constraints as above. Briefly, we identify some x = replaceAll(y, e, z) where x does not appear as an argument to any other use of replaceAll. We then transform any regular constraints on x into additional constraints on y and z. This allows us to remove the variable x since the extended constraints on y and z are sufficient for determining satisfiability. Moreover, from a satisfying assignment to y and z we can construct a satisfying assignment to x as well. This is the technical part of our decision procedure and is explained in detail in the following sections, for increasingly complex uses of replaceAll.

6 DECISION PROCEDURE FOR SL[replaceAll]: THE SINGLE-LETTER CASE

In this section, we consider the single-letter case, that is, for the SL[replaceAll] formula $C = \varphi \wedge \psi$, every term of the form replaceAll(z, e, z') in φ satisfies that $e = a$ for $a \in \Sigma$. We begin by explaining the idea of the decision procedure in the case where there is a single use of a replaceAll$(-, -, -)$ term. Then we describe the decision procedure in full details.

6.1 A Single Use of replaceAll$(-, -, -)$

Let us start with the simple case that

$$C \equiv x = \text{replaceAll}(y, a, z) \wedge x \in e_1 \wedge y \in e_2 \wedge z \in e_3,$$

where, for $i = 1, 2, 3$, we suppose $\mathcal{A}_i = (Q_i, \delta_i, q_{0,i}, F_i)$ is the NFA corresponding to the regular expression e_i.

From the semantics, C is satisfiable if and only if x, y, z can be assigned with strings u, v, w so that: (1) u is obtained from v by replacing all the occurrences of a in v with w, and (2) u, v, w are accepted by $\mathcal{A}_1, \mathcal{A}_2, \mathcal{A}_3$ respectively. Let u, v, w be the strings satisfying these two constraints. As u is accepted by $\mathcal{A}_1$, there must be an accepting run of $\mathcal{A}_1$ on u. Let $v = v_1 a v_2 a \cdots a v_k$ such that for each $i \in [k]$, $v_i \in (\Sigma \setminus \{a\})^*$. Then $u = v_1 w v_2 w \cdots w v_k$ and there are states $q_1, q'_1, \cdots, q_{k-1}, q'_{k-1}, q_k$ such that

$$q_{0,1} \xrightarrow[\mathcal{A}_1]{v_1} q_1 \xrightarrow[\mathcal{A}_1]{w} q'_1 \xrightarrow[\mathcal{A}_1]{v_2} q_2 \xrightarrow[\mathcal{A}_1]{w} q'_2 \cdots q_{k-1} \xrightarrow[\mathcal{A}_1]{w} q'_{k-1} \xrightarrow[\mathcal{A}_1]{v_k} q_k$$

and $q_k \in F_1$. Let T_z denote $\{(q_i, q'_i) \mid i \in [k-1]\}$. Then $w \in \mathcal{L}(\mathcal{A}_3) \cap \bigcap\limits_{(q,q') \in T_z} \mathcal{L}(\mathcal{A}_1(q, q'))$. In addition, let $\mathcal{B}_{\mathcal{A}_1, a, T_z}$ be the NFA obtained from $\mathcal{A}_1$ by removing all the a-transitions first and then adding the a-transitions (q, a, q') for $(q, q') \in T_z$. Then

$$q_{0,1} \xrightarrow[\mathcal{B}_{\mathcal{A}_1,a,T_z}]{v_1} q_1 \xrightarrow[\mathcal{B}_{\mathcal{A}_1,a,T_z}]{a} q'_1 \xrightarrow[\mathcal{B}_{\mathcal{A}_1,a,T_z}]{v_2} q_2 \xrightarrow[\mathcal{B}_{\mathcal{A}_1,a,T_z}]{a} q'_2 \cdots q_{k-1} \xrightarrow[\mathcal{B}_{\mathcal{A}_1,a,T_z}]{a} q'_{k-1} \xrightarrow[\mathcal{B}_{\mathcal{A}_1,a,T_z}]{v_k} q_k.$$

Therefore, $v \in \mathcal{L}(\mathcal{A}_2) \cap \mathcal{L}(\mathcal{B}_{\mathcal{A}_1,a,T_z})$. We deduce that there is $T_z \subseteq Q_1 \times Q_1$ such that $\mathcal{L}(\mathcal{A}_3) \cap \bigcap\limits_{(q,q') \in T_z} \mathcal{L}(\mathcal{A}_1(q, q')) \neq \emptyset$ and $\mathcal{L}(\mathcal{A}_2) \cap \mathcal{L}(\mathcal{B}_{\mathcal{A}_1,a,T_z}) \neq \emptyset$. In addition, it is not hard to see that this condition is also sufficient for the satisfiability of C. The arguments proceed as follows: Let $v \in \mathcal{L}(\mathcal{A}_2) \cap \mathcal{L}(\mathcal{B}_{\mathcal{A}_1,a,T_z})$ and $w \in \mathcal{L}(\mathcal{A}_3) \cap \bigcap\limits_{(q,q') \in T_z} \mathcal{L}(\mathcal{A}_1(q, q'))$. From $v \in \mathcal{L}(\mathcal{B}_{\mathcal{A}_1,a,T_z})$, we know that there is an accepting run of $\mathcal{B}_{\mathcal{A}_1,a,T_z}$ on v. Recall that $\mathcal{B}_{\mathcal{A}_1,a,T_z}$ is obtained from $\mathcal{A}_1$ by first removing all the a-transitions, then adding all the transitions (q, a, q') for $(q, q') \in T_z$. Suppose $v = v_1 a v_2 \cdots a v_k$ such that $v_i \in (\Sigma \setminus \{a\})^*$ for each $i \in [k]$ and

$$q_{0,1} \xrightarrow[\mathcal{B}_{\mathcal{A}_1,a,T_z}]{v_1} q_1 \xrightarrow[\mathcal{B}_{\mathcal{A}_1,a,T_z}]{a} q'_1 \xrightarrow[\mathcal{B}_{\mathcal{A}_1,a,T_z}]{v_2} q_2 \xrightarrow[\mathcal{B}_{\mathcal{A}_1,a,T_z}]{a} q'_2 \cdots q_{k-1} \xrightarrow[\mathcal{B}_{\mathcal{A}_1,a,T_z}]{a} q'_{k-1} \xrightarrow[\mathcal{B}_{\mathcal{A}_1,a,T_z}]{v_k} q_k$$

is an accepting run of $\mathcal{B}_{\mathcal{A}_1,a,T_z}$ on v. Then $q_{0,1} \xrightarrow[\mathcal{A}_1]{v_1} q_1$, and for each $i \in [k-1]$ we have $(q_i, q'_i) \in T_z$ and $q'_i \xrightarrow[\mathcal{A}_1]{v_{i+1}} q_{i+1}$; moreover, $q_k \in F_1$. Let $u = \mathsf{replaceAll}(v, a, w) = v_1 w v_2 \cdots w v_k$. Since $w \in \bigcap\limits_{(q,q') \in T_z} \mathcal{L}(\mathcal{A}_1(q, q'))$, we infer that

$$q_{0,1} \xrightarrow[\mathcal{A}_1]{v_1} q_1 \xrightarrow[\mathcal{A}_1]{w} q'_1 \xrightarrow[\mathcal{A}_1]{v_2} q_2 \xrightarrow[\mathcal{A}_1]{w} q'_2 \cdots q_{k-1} \xrightarrow[\mathcal{A}_1]{w} q'_{k-1} \xrightarrow[\mathcal{A}_1]{v_k} q_k$$

is an accepting run of $\mathcal{A}_1$ on u. Therefore, u is accepted by $\mathcal{A}_1$ and C is satisfiable.

Proposition 6.1. *We have $C \equiv x = \mathsf{replaceAll}(y, a, z) \wedge x \in e_1 \wedge y \in e_2 \wedge z \in e_3$ is satisfiable iff there exists $T_z \subseteq Q_1 \times Q_1$ with $\mathcal{L}(\mathcal{A}_3) \cap \bigcap\limits_{(q,q') \in T_z} \mathcal{L}(\mathcal{A}_1(q, q')) \neq \emptyset$ and $\mathcal{L}(\mathcal{A}_2) \cap \mathcal{L}(\mathcal{B}_{\mathcal{A}_1,a,T_z}) \neq \emptyset$.*

From Proposition 6.1, we can decide the satisfiability of C in polynomial space as follows:

Step I. Nondeterministically choose a set $T_z \subseteq Q_1 \times Q_1$.
Step II. Nondeterministically choose an accepting run of the product automaton of $\mathcal{A}_3$ and $\mathcal{A}_1(q, q')$ for $(q, q') \in T_z$.
Step III. Nondeterministically choose an accepting run of the product automaton of $\mathcal{A}_2$ and $\mathcal{B}_{\mathcal{A}_1,a,T_z}$.

During Step II and III, it is sufficient to record T_z and a state of the product automaton, which occupies only a polynomial space.

The above decision procedure can be easily generalised to the case that there are multiple atomic regular constraints for x. For instance, let $x \in e_{1,1} \wedge x \in e_{1,2}$ and for $j = 1, 2$, $\mathcal{A}_{1,j} = (Q_{1,j}, \delta_{1,j}, q_{0,1,j}, F_{1,j})$ be the NFA corresponding to $e_{1,j}$. Then in Step I, two sets $T_{1,z} \subseteq Q_{1,1} \times Q_{1,1}$ and $T_{2,z} \subseteq Q_{1,2} \times Q_{1,2}$ are nondeterministically chosen, moreover, Step II and III are adjusted accordingly.

Example 6.2. Let $C \equiv x = \mathsf{replaceAll}(y, 0, z) \wedge x \in e_1 \wedge y \in e_2 \wedge z \in e_3$, where $e_1 = (0 + 1)^*(00(0 + 1)^* + 11(0 + 1)^*)$, $e_2 = (01)^*$, and $e_3 = (10)^*$. The NFA $\mathcal{A}_1, \mathcal{A}_2, \mathcal{A}_3$ corresponding to e_1, e_2, e_3 respectively are illustrated in Figure 2. Let $T_z = \{(q_0, q_0), (q_1, q_2)\}$. Then

$$\begin{array}{rcl} \mathcal{L}(\mathcal{A}_3) \cap \bigcap\limits_{(q,q')\in T_z} \mathcal{L}(\mathcal{A}_1(q,q')) & = & \mathcal{L}(\mathcal{A}_3) \cap \mathcal{L}(\mathcal{A}_1(q_0,q_0)) \cap \mathcal{L}(\mathcal{A}_1(q_1,q_2)) \\ & = & \mathcal{L}((10)^*) \cap \mathcal{L}((0+1)^*) \cap \mathcal{L}(1(0+1)^*) \\ & \neq & \emptyset. \end{array}$$

In addition, $\mathcal{B}_{\mathcal{A}_1,0,T_z}$ (also illustrated in Figure 2) is obtained from $\mathcal{A}_1$ by removing all the 0-transitions, then adding the transitions $(q_0, 0, q_0)$ and $(q_1, 0, q_2)$. Then

$$\mathcal{L}(\mathcal{A}_2) \cap \mathcal{L}(\mathcal{B}_{\mathcal{A}_1,0,T_z}) = \mathcal{L}((01)^*) \cap \mathcal{L}((0+1)^*101^*) \neq \emptyset.$$

We can choose z to be a string from $\mathcal{L}(\mathcal{A}_3) \cap \bigcap\limits_{(q,q')\in T_z} \mathcal{L}(\mathcal{A}_1(q,q')) = \mathcal{L}((10)^*) \cap \mathcal{L}((0+1)^*) \cap \mathcal{L}(1(0+1)^*)$, say 10, and y to be a string from $\mathcal{L}(\mathcal{A}_2) \cap \mathcal{L}(\mathcal{B}_{\mathcal{A}_1,0,T_z}) = \mathcal{L}((01)^*) \cap \mathcal{L}((0+1)^*101^*)$, say 0101, then we set x to $\mathsf{replaceAll}(0101, 0, 10) = 101101$, which is in $\mathcal{L}(\mathcal{A}_1)$. Thus, C is satisfiable. □

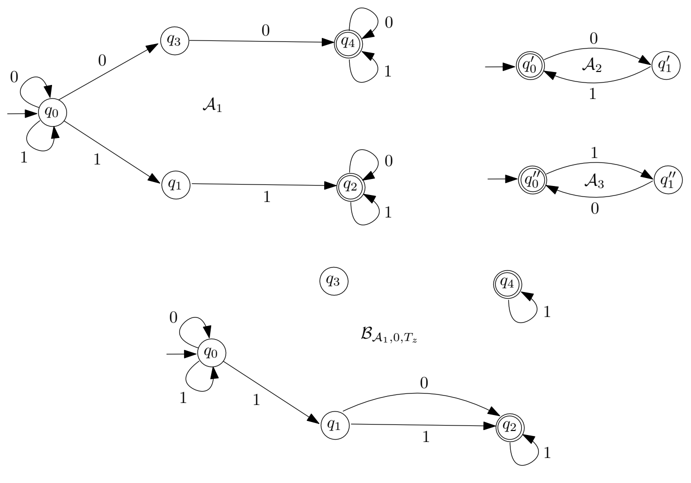


Fig. 2. An example for the single-letter case: One replaceAll

6.2 The General Case

Let us now consider the general case where C contains multiple occurrences of $\mathsf{replaceAll}(-, -, -)$ terms. Then the satisfiability of C is decided by the following two-step procedure.

Step I. We utilise the dependency graph C and compute nondeterministically a collection of atomic regular constraints $\mathcal{E}(x)$ for each variable x, in a top-down manner.

Notice that $\mathcal{E}(x)$ is represented succinctly as a set of pairs $(\mathcal{T}, \mathcal{P})$, where $\mathcal{T} = (Q, \delta)$ is a transition graph and $\mathcal{P} \subseteq Q \times Q$. The intention of $(\mathcal{T}, \mathcal{P})$ is to represent succinctly the collection of the atomic regular constraints containing $(Q, \delta, q, \{q'\})$ for each $(q, q') \in \mathcal{P}$, where q is the initial state and $\{q'\}$ is the set of final states.

Initially, let $G_0 := G_C$. In addition, for each variable x, we define $\mathcal{E}_0(x)$ as follows: Let $x \in e_1 \wedge \cdots \wedge x \in e_n$ be the conjunction of all the atomic regular constraints related to x in C. For each $i \in [n]$, let $\mathcal{A}_i = (Q_i, \delta_i, q_{0,i}, F_i)$ be the NFA corresponding to e_i. We nondeterministically choose $q_i \in F_i$ and set $\mathcal{E}_0(x) := \big\{((Q_i, \delta_i), \{(q_{0,i}, q_i)\}) \mid i \in [n]\big\}$.

We begin with $i := 0$ and repeat the following procedure until we reach some i where G_i is an empty graph, i.e. a graph without edges. Note that G_0 was defined above.

(1) Select a vertex x of G_i such that x has no predecessors and has two successors via edges $(x, (\mathsf{l}, a), y)$ and $(x, (\mathsf{r}, a), z)$ in G_i. Suppose $\mathcal{E}_i(x) = \{(\mathcal{T}_1, \mathcal{P}_1), \cdots, (\mathcal{T}_k, \mathcal{P}_k)\}$, where for each $j \in [k]$, $\mathcal{T}_j = (Q_j, \delta_j)$. Then $\mathcal{E}_{i+1}(z)$ and $\mathcal{E}_{i+1}(y)$ and G_{i+1} are computed as follows:
   (a) For each $j \in [k]$, nondeterministically choose a set $T_{j,z} \subseteq Q_j \times Q_j$.
   (b) If $y \neq z$, then let

$$\mathcal{E}_{i+1}(z) := \mathcal{E}_i(z) \cup \big\{(\mathcal{T}_j, T_{j,z}) \mid j \in [k]\big\} \text{ and } \mathcal{E}_{i+1}(y) := \mathcal{E}_i(y) \cup \big\{(\mathcal{T}_{\mathcal{T}_j, a, T_{j,z}}, \mathcal{P}_j) \mid j \in [k]\big\}$$

   where $\mathcal{T}_{\mathcal{T}_j, a, T_{j,z}}$ is obtained from $\mathcal{T}_j$ by first removing all the a-transitions, then adding all the transitions (q, a, q') for $(q, q') \in T_{j,z}$. Otherwise, let $\mathcal{E}_{i+1}(z) := \mathcal{E}_i(z) \cup \big\{(\mathcal{T}_j, T_{j,z}) \mid j \in [k]\big\} \cup \big\{(\mathcal{T}_{\mathcal{T}_j, a, T_{j,z}}, \mathcal{P}_j) \mid j \in [k]\big\}$. In addition, for each vertex x' distinct from y, z, let $\mathcal{E}_{i+1}(x') := \mathcal{E}_i(x')$.
   (c) Let $G_{i+1} := G_i \setminus \{(x, (\mathsf{l}, a), y), (x, (\mathsf{r}, a), z)\}$.
(2) Let $i := i + 1$.

For each variable x, let $\mathcal{E}(x)$ denote the set $\mathcal{E}_i(x)$ after exiting the above loop.

Step II. Output "satisfiable" if for each source variable x there is an accepting run of the product of all the NFA in $\mathcal{E}(x)$; otherwise, output "unsatisfiable".

It remains to argue the correctness and complexity of the above procedure and show how to obtain satisfying assignments to satisfiable constraints. Correctness follows a similar argument to Proposition 6.1 and is presented in the full version. Intuitively, Proposition 6.1 shows our procedure correctly eliminates occurrences of replaceAll until only regular constraints remain.

If, in the case that the equation is satisfiable, one wishes to obtain a satisfying assignment to all variables, we can proceed as follows. First, for each source variable x, nondeterministically choose an accepting run of the product of all the NFA in $\mathcal{E}(x)$. As argued in the full version, the word labelling this run satisfies all regular constraints on x since it is taken from a language that is guaranteed to be a subset of the set of words satisfying the original constraints. For non-source variables, we derive an assignment as in Proposition 6.1, proceeding by induction from the source variables. That is, select some variable x such that x is derived from variables y and z and assignments to both y and z have already been obtained. The value for x is immediately obtained by performing the replaceAll operation using the assignments to y and z. That this value satisfies all regular constraints on x follows the same argument as Proposition 6.1. The procedure terminates when all variables have been assigned.

We now give an example before proceeding to the complexity analysis.

Example 6.3. Suppose $C \equiv x = \mathsf{replaceAll}(y, 0, z) \wedge y = \mathsf{replaceAll}(y', 1, z') \wedge x \in e_1 \wedge y \in e_2 \wedge z \in e_3 \wedge y' \in e_4 \wedge z' \in e_5$, where e_1, e_2, e_3 are as in Example 6.2, $e_4 = 0^*1^*0^*1^*$, and $e_5 = 0^*1^*$. Let $\mathcal{A}_4, \mathcal{A}_5$ be the NFA corresponding to e_4 and e_5 respectively (see Figure 3). The dependency graph G_C of C is illustrated in Figure 3. Let $\mathcal{T}_1, \cdots, \mathcal{T}_5$ be the transition graph of $\mathcal{A}_1, \cdots, \mathcal{A}_5$ respectively. Then the collection of regular constraints $\mathcal{E}(\cdot)$ are computed as follows.

- Let $G_0 = G_C$. Pick the sets $\mathcal{E}_0(x) = \{(\mathcal{T}_1, \{(q_0, q_2)\})\}$, $\mathcal{E}_0(y) = \{(\mathcal{T}_2, \{(q'_0, q'_0)\})\}$, $\mathcal{E}_0(z) = \{(\mathcal{T}_3, \{(q''_0, q''_0)\})\}$, $\mathcal{E}_0(y') = \{(\mathcal{T}_4, \{(p_0, p_1)\})\}$, and $\mathcal{E}_0(z') = \{(\mathcal{T}_5, \{(p'_0, p'_1)\})\}$ nondeterministically.
- Select the vertex x in G_0, construct $\mathcal{E}_1(y)$ and $\mathcal{E}_1(z)$ as in Example 6.2, that is, nondeterministically choose $T_z = \{(q_0, q_0), (q_1, q_2)\}$, let

$$\mathcal{E}_1(z) = \{(\mathcal{T}_3, \{(q''_0, q''_0)\}), (\mathcal{T}_1, \{(q_0, q_0), (q_1, q_2)\})\} \text{ and } \mathcal{E}_1(y) = \{(\mathcal{T}_2, \{(q'_0, q'_0)\}), (\mathcal{T}_{\mathcal{T}_1,0,T_z}, \{(q_0, q_2)\})\},$$

  where $\mathcal{T}_{\mathcal{T}_1,0,T_z}$ is the transition graph of $\mathcal{B}_{\mathcal{A}_1,0,T_z}$ illustrated in Figure 2. In addition, $\mathcal{E}_1(x) = \mathcal{E}_0(x)$, $\mathcal{E}_1(y') = \mathcal{E}_0(y')$ and $\mathcal{E}_1(z') = \mathcal{E}_0(z')$. Finally, we get G_1 from G_0 by removing the two edges from x.
- Select the vertex y in G_1, construct $\mathcal{E}_2(y')$ and $\mathcal{E}_2(z')$ as follows: Nondeterministically choose $T_{1,z'} = \{(q'_0, q'_0)\}$ for $\mathcal{T}_2$ and $T_{2,z'} = \{(q_0, q_1), (q_1, q_2)\}$ for $\mathcal{T}_{\mathcal{T}_1,0,T_z}$, let

$$\mathcal{E}_2(z') = \left\{(\mathcal{T}_5, \{(p'_0, p'_1)\}), (\mathcal{T}_2, \{(q'_0, q'_0)\}), (\mathcal{T}_{\mathcal{T}_1,0,T_z}, \{(q_0, q_1), (q_1, q_2)\})\right\}, \text{ and}$$

$$\mathcal{E}_2(y') = \left\{(\mathcal{T}_4, \{(p_0, p_1)\}), (\mathcal{T}_{\mathcal{T}_2,1,T_{1,z'}}, \{(q'_0, q'_0)\}), (\mathcal{T}_{\mathcal{T}_{\mathcal{T}_1,0,T_z},1,T_{2,z'}}, \{(q_0, q_2)\})\right\},$$

  where $\mathcal{T}_{\mathcal{T}_2,1,T_{1,z'}}$ and $\mathcal{T}_{\mathcal{T}_{\mathcal{T}_1,0,T_z},1,T_{2,z'}}$ are shown in Figure 4. In addition, $\mathcal{E}_2(x) = \mathcal{E}_1(x)$, $\mathcal{E}_2(y) = \mathcal{E}_1(y)$, and $\mathcal{E}_2(z) = \mathcal{E}_1(z)$. Finally, we get G_2 from G_1 by removing the two edges from y.

Since G_2 contains no edges, we have $\mathcal{E}(x) = \mathcal{E}_2(x)$, similarly for $\mathcal{E}(y)$, $\mathcal{E}(z)$, $\mathcal{E}(y')$, and $\mathcal{E}(z')$. For the three source variables y', z', z, it is not hard to check that 01 belongs to the intersection of the regular constraints in $\mathcal{E}(z')$, 11 belongs to the intersection of the regular constraints in $\mathcal{E}(y')$, and 10 belongs to the intersection of the regular constraints in $\mathcal{E}(z)$. Then y takes the value replaceAll$(11, 1, 01) = 0101 \in \mathcal{L}(e_2)$, and x takes the value replaceAll$(0101, 0, 10) = 101101 \in \mathcal{L}(e_1)$. Therefore, C is satisfiable. □

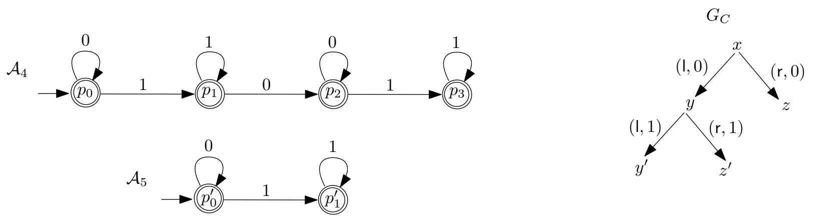


Fig. 3. An example for the single-letter case: Multiple replaceAll

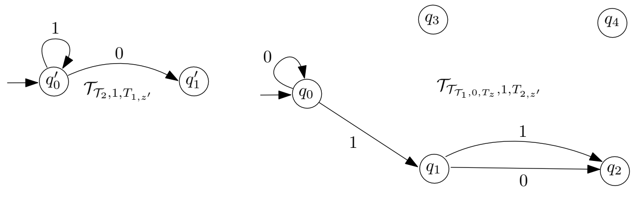

q'_0
q'_1
$\mathcal{T}_{\mathcal{T}_2,1,T_{1,z'}}$
q_0
q_1
q_2
q_3
q_4
$\mathcal{T}_{\mathcal{T}_{\mathcal{T}_1,0,T_z},1,T_{2,z'}}$

Fig. 4. $\mathcal{T}_{\mathcal{T}_2,1,T_{1,z'}}$ and $\mathcal{T}_{\mathcal{T}_{\mathcal{T}_1,0,T_z},1,T_{2,z'}}$

6.2.1 Complexity. To show our decision procedure works in exponential space, it is sufficient to show that the cardinalities of the sets $\mathcal{E}(x)$ are exponential w.r.t. the size of C.

Proposition 6.4. *The cardinalities of $\mathcal{E}(x)$ for the variables x in G_C are at most exponential in $\mathsf{Idx}_{\mathsf{dmd}}(G_C)$, the diamond index of G_C.*

Therefore, according to Proposition 6.4, if the diamond index of G_C is bounded by a constant c, then the cardinalities of $\mathcal{E}(x)$ become *polynomial* in the size of C and we obtain a polynomial space decision procedure. In this case, we conclude that the satisfiability problem is PSPACE-complete.

Proof of Proposition 6.4. Let K be the maximum of $|\mathcal{E}_0(x)|$ for $x \in \mathsf{Vars}(\varphi)$. For each variable x in G_C, all the regular constraints in $\mathcal{E}(x)$ are either from $\mathcal{E}_0(x)$, or are generated from some regular constraints from $\mathcal{E}_0(x')$ for the ancestors x' of x. Let x' be an ancestor of x. Then for each $(\mathcal{T}, \mathcal{P}) \in \mathcal{E}_0(x')$, according to Step I in the decision procedure, by an induction on the maximum length of the paths in from x' to x, we can show that the number of elements in $\mathcal{E}(x)$ that are generated from $(\mathcal{T}, \mathcal{P})$ is at most the number of different paths from x' to x. From Proposition 4.5, we know that there are at most $(|\mathsf{Vars}(\varphi)| \cdot |E_C|)^{O(\mathsf{Idx}_{\mathsf{dmd}}(G_C))}$ different paths from x' to x. Since there are at most $|\mathsf{Vars}(\varphi)|$ ancestors of x, we deduce that $|\mathcal{E}(x)| \leq K \cdot |\mathsf{Vars}(\varphi)| \cdot (|\mathsf{Vars}(\varphi)||E_C|)^{O(\mathsf{Idx}_{\mathsf{dmd}}(G_C))}$. □

7 DECISION PROCEDURE FOR SL[replaceAll]: THE CONSTANT-STRING CASE

In this section, we consider the constant-string special case, that is, for an SL[replaceAll] formula $C = \varphi \wedge \psi$, every term of the form $\mathsf{replaceAll}(z, e, z')$ in φ satisfies that $e = u$ for $u \in \Sigma^+$. Note that the case when $u = \epsilon$ will be dealt with in Section 8.

Again, let us start with the simple situation that $C \equiv x = \mathsf{replaceAll}(y, u, z) \wedge x \in e_1 \wedge y \in e_2 \wedge z \in e_3$, where $|u| \geq 2$. For $i = 1, 2, 3$, let $\mathcal{A}_i = (Q_i, \delta_i, q_{0,i}, F_i)$ be the NFA corresponding to e_i. In addition, let $k = |u|$ and $u = a_1 \cdots a_k$ with $a_i \in \Sigma$ for each $i \in [k]$.

From the semantics, C is satisfiable iff x, y, z can be assigned with strings v, w, w' such that: (1) $v = \mathsf{replaceAll}(w, u, w')$, and (2) v, w, w' are accepted by $\mathcal{A}_1, \mathcal{A}_2, \mathcal{A}_3$ respectively. Let v, w, w' be the strings satisfying these two constraints. Since $v = \mathsf{replaceAll}(w, u, w')$, we know that there are strings $w_1, w_2, \cdots, w_n$ such that $w = w_1 u w_2 \cdots u w_n$ and $v = w_1 w' w_2 \cdots w' w_n$. As v is accepted by $\mathcal{A}_1$, there is an accepting run of $\mathcal{A}_1$ on v, say

$$q_{0,1} \xrightarrow[\mathcal{A}_1]{w_1} q_1 \xrightarrow[\mathcal{A}_1]{w'} q_1' \xrightarrow[\mathcal{A}_1]{w_2} q_2 \xrightarrow[\mathcal{A}_1]{w'} q_2' \cdots q_{n-1} \xrightarrow[\mathcal{A}_1]{w'} q_{n-1}' \xrightarrow[\mathcal{A}_1]{w_n} q_n.$$

Let $T_z = \{(q_i, q_i') \mid i \in [n]\}$. Then $w' \in \mathcal{L}(\mathcal{A}_3) \cap \bigcap_{(q,q') \in T_z} \mathcal{L}(\mathcal{A}_1(q, q'))$. Therefore, $\mathcal{L}(\mathcal{A}_3) \cap \bigcap_{(q,q') \in T_z} \mathcal{L}(\mathcal{A}_1(q, q')) \neq \emptyset$. Similar to the single-letter case, we construct an NFA $\mathcal{B}_{\mathcal{A}_1, u, T_z}$ to characterise the satisfiability of C. More precisely, C is satisfiable iff there is $T_z \subseteq Q_1 \times Q_1$ such that $\mathcal{L}(\mathcal{A}_3) \cap \bigcap_{(q,q') \in T_z} \mathcal{L}(\mathcal{A}_1(q, q')) \neq \emptyset$ and $\mathcal{L}(\mathcal{A}_2) \cap \mathcal{L}(\mathcal{B}_{\mathcal{A}_1, u, T_z}) \neq \emptyset$. Intuitively, when reading the string w, $\mathcal{B}_{\mathcal{A}_1, u, T_z}$ simulates the generation of v from w and w' (that is, the replacement of every occurrence of u in w with w') and verifies that v is accepted by $\mathcal{A}_1$, by using T_z. To build $\mathcal{B}_{\mathcal{A}_1, u, T_z}$, we utilise the concepts of window profiles and parsing automata defined below. Intuitively, a window profile keeps track of which positions in the preceding characters could form the beginning of a match of u.

Definition 7.1 (window profiles w.r.t. u). Let v be a nonempty string with $k = |v|$, and $i \in [k]$. Then the *window profile of the position i in v w.r.t. u* is $\overrightarrow{W} \in \{\bot, \top\}^{k-1}$ defined as follows:

- If $i \geq k - 1$, then for each $j \in [k-1]$, $\overrightarrow{W}[j] = \top$ iff $v[i - j + 1] \cdots v[i] = u[1] \cdots u[j]$.

- If $i < k-1$, then for each $j \in [i]$, $\overrightarrow{W}[j] = \top$ iff $v[i-j+1]\cdots v[i] = u[1]\cdots u[j]$, and for each $j : i < j \leq k-1$, $\overrightarrow{W}[j] = \bot$.

Let WP_u denote the set of window profiles of the positions in nonempty strings w.r.t. u.

Proposition 7.2. $|\mathsf{WP}_u| \leq |u|$.

Proof. Let $k = |u|$. For each profile $\overrightarrow{W}$, let v be a nonempty string and i be a position of v such that for each $j \in [k-1]$, $\overrightarrow{W}[j] = \top$ iff $v[i-j+1]\dots v[i] = u[1]\dots u[j]$. Define $\mathsf{idx}_{\overrightarrow{W}}$ as follows: If there is $j \in [k-1]$ such that $\overrightarrow{W}[j] = \top$, then $\mathsf{idx}_{\overrightarrow{W}}$ is the maximum of such indices $j \in [k-1]$, otherwise, $\mathsf{idx}_{\overrightarrow{W}} = 0$. The following fact holds for $\overrightarrow{W}$ and $\mathsf{idx}_{\overrightarrow{W}}$:

- for each $j' : \mathsf{idx}_{\overrightarrow{W}} < j' \leq k-1$, $\overrightarrow{W}[j'] = \bot$,
- in addition, since $v[i-\mathsf{idx}_{\overrightarrow{W}}+1]\cdots v[i] = u[1]\cdots u[\mathsf{idx}_{\overrightarrow{W}}]$, the values of $\overrightarrow{W}[1], \cdots, \overrightarrow{W}[\mathsf{idx}_{\overrightarrow{W}}]$ are completely determined by $u[1]\cdots u[\mathsf{idx}_{\overrightarrow{W}}]$.

Let $\eta : \mathsf{WP}_u \to \{0\} \cup [k-1]$ be a function such that for each $\overrightarrow{W} \in \mathsf{WP}_u$, $\eta(\overrightarrow{W}) = \mathsf{idx}_{\overrightarrow{W}}$. Then η is an injective function, since for every $\overrightarrow{W}, \overrightarrow{W'} \in \mathsf{WP}_u$, $\mathsf{idx}_{\overrightarrow{W}} = \mathsf{idx}_{\overrightarrow{W'}}$ iff $\overrightarrow{W} = \overrightarrow{W'}$. Therefore, we conclude that $|\mathsf{WP}_u| \leq k$. □

Example 7.3. Let $\Sigma = \{0,1\}$, $u = 010$. Then $\mathsf{WP}_u = \{\bot\bot, \top\bot, \bot\top\}$.

- Consider the string $v = 1$ and the position $i = 1$ in v. Since $v[1] = 1 \neq u[1] = 0$, the window profile of i in v w.r.t. u is $\bot\bot$.
- Consider the string $v = 00$ and the position $i = 2$ in v. Since $v[2] = u[1]$ and $v[1]v[2] \neq u[1]u[2]$, the window profile of i in v w.r.t. u is $\top\bot$.
- Consider the string $v = 01$ and the position $i = 2$ in v. Since $v[2] \neq u[1]$ and $v[1]v[2] = u[1]u[2]$, the window profile of i in v w.r.t. u is $\bot\top$.

Note that $\top\top \notin \mathsf{WP}_u$, since for every string v and the position i in v, if $v[i-1]v[i] = u[1]u[2] = 01$, then $v[i] = 1 \neq 0 = u[1]$.

We will construct a parsing automaton $\mathcal{A}_u$ from u, which parses a string v containing at least one occurrence of u (i.e. $v \in \Sigma^* u \Sigma^*$) into $v_1 u v_2 u \dots v_l u v_{l+1}$ such that $v_j u[1]\dots u[k-1] \notin \Sigma^* u \Sigma^*$ for each $1 \leq j \leq l$. This ensures that the only occurrence of u in each $v_j u$ is a suffix. Finally, we also require $v_{l+1} \notin \Sigma^* u \Sigma^*$. The window profiles w.r.t. u will be used to ensure that v is correctly parsed, namely, the first, second, $\cdots$, occurrences of u are correctly identified.

Definition 7.4 (Parsing automata). Given a string u we define the *parsing automaton* $\mathcal{A}_u$ to be the NFA $(Q_u, \delta_u, q_{0,u}, F_u)$ where $q_{0,u} = q_0$ and the remaining components are given below.

- $Q_u = \{q_0\} \cup \left\{\left(\mathsf{search}, \overrightarrow{W}\right) \middle| \overrightarrow{W} \in \mathsf{WP}_u\right\} \cup \left\{\left(\mathsf{vfy}, j, \overrightarrow{W}\right) \middle| j \in [k-1], \overrightarrow{W} \in \mathsf{WP}_u\right\}$, where q_0 is a distinguished state whose purpose will become clear later on, and the tags "search" and "vfy" are used to denote whether $\mathcal{A}_u$ is in the "search" mode to search for the next occurrence of u, or in the "verify" mode to verify that the current position is a part of an occurrence of u.
- δ_u is defined as follows.
  - The transition $\left(q_0, a, \left(\mathsf{search}, \overrightarrow{W}\right)\right) \in \delta_u$, where $\overrightarrow{W}[1] = \top$ iff $a = u[1]$, and for each $i : 2 \leq i \leq k-1$, $\overrightarrow{W}[i] = \bot$.
  - The transition $\left(q_0, u[1], \left(\mathsf{vfy}, 1, \overrightarrow{W}\right)\right) \in \delta_u$, where $\overrightarrow{W}[1] = \top$ and for each $i : 2 \leq i \leq k-1$, $\overrightarrow{W}[i] = \bot$.

- For each state $\left(\text{search}, \overrightarrow{W}\right)$ and $a \in \Sigma$ such that $\overrightarrow{W}[k-1] = \bot$ or $a \neq u[k]$,
  - the transition $\left(\left(\text{search}, \overrightarrow{W}\right), a, \left(\text{search}, \overrightarrow{W'}\right)\right) \in \delta_u$, where $\overrightarrow{W'}[1] = \top$ iff $a = u[1]$, and for each $i : 2 \leq i \leq k-1$, $\overrightarrow{W'}[i] = \top$ iff ($\overrightarrow{W}[i-1] = \top$ and $a = u[i]$),
  - if $a = u[1]$, then the transition $\left(\left(\text{search}, \overrightarrow{W}\right), a, \left(\text{vfy}, 1, \overrightarrow{W'}\right)\right) \in \delta_u$, where $\overrightarrow{W'}[1] = \top$, and for each $i : 2 \leq i \leq k-1$, $\overrightarrow{W'}[i] = \top$ iff ($\overrightarrow{W}[i-1] = \top$ and $a = u[i]$).
- For each state $\left(\text{vfy}, i-1, \overrightarrow{W}\right)$ and $a \in \Sigma$ such that
  - $2 \leq i \leq k-1$,
  - $\overrightarrow{W}[i-1] = \top$, $a = u[i]$, and
  - either $\overrightarrow{W}[k-1] = \bot$ or $a \neq u[k]$,

  we have $\left(\left(\text{vfy}, i-1, \overrightarrow{W}\right), a, \left(\text{vfy}, i, \overrightarrow{W'}\right)\right) \in \delta_u$, where for each $j : 2 \leq j \leq k-1$, $\overrightarrow{W'}[j] = \top$ iff $\overrightarrow{W}[j-1] = \top$ and $a = u[j]$.
- For each state $\left(\text{vfy}, k-1, \overrightarrow{W}\right)$ and $a \in \Sigma$ such that $\overrightarrow{W}[k-1] = \top$ and $a = u[k]$, we have $\left(\left(\text{vfy}, k-1, \overrightarrow{W}\right), a, q_0\right) \in \delta_u$.

Note that the constraint $\overrightarrow{W}[k-1] = \bot$ or $a \neq u[k]$ is used to guarantee that each occurrence of the state q_0, except the first one, witnesses the *first* occurrence of u from the beginning or after its previous occurrence. In other words, the constraint $\overrightarrow{W}[k-1] = \bot$ or $a \neq u[k]$ is used to guarantee that after an occurrence of q_0, if q_0 has not been reached again, then u is forbidden to occur.

• $F_u = \{q_0\} \cup \left\{\left(\text{search}, \overrightarrow{W}\right) \mid \overrightarrow{W} \in \text{WP}_u\right\}$.

Note that the states $\left(\text{vfy}, j, \overrightarrow{W}\right)$ are not final states, since, when in these states, the verification of the current occurrence of u has not been complete yet.

Let $Q_{\text{search}} = \left\{\left(\text{search}, \overrightarrow{W}\right) \mid \overrightarrow{W} \in \text{WP}_u\right\}$, and $Q_{\text{vfy},i} = \left\{\left(\text{vfy}, i, \overrightarrow{W}\right) \mid \overrightarrow{W} \in \text{WP}_u\right\}$ for each $i \in [k-1]$. In addition, let $Q_{\text{vfy}} = \bigcup\limits_{i \in [k-1]} Q_{\text{vfy},i}$. Suppose $v = v_1 u v_2 u \cdots v_l u v_{l+1}$ such that $v_j u[1] \ldots u[k-1] \notin \Sigma^* u \Sigma^*$ for each $1 \leq j \leq l$, in addition, $v_{l+1} \notin \Sigma^* u \Sigma^*$. Then there exists a *unique* accepting run r of $\mathcal{A}_u$ on v such that the state sequence in r is of the form $q_0\ r_1\ q_0\ r_2\ q_0\ \cdots\ r_l\ q_0\ r_{l+1}$, where for each $j \in [l]$, $r_j \in \mathcal{L}((Q_{\text{search}})^+ \circ Q_{\text{vfy},1} \circ \cdots \circ Q_{\text{vfy},k-1})$, and $r_{l+1} \in \mathcal{L}((Q_{\text{search}})^*)$.

Example 7.5. Consider $u = 010$ in Example 7.3. The parsing automaton $\mathcal{A}_u$ is illustrated in Figure 5. Note that there are no 0-transitions out of $(\text{search}, \bot\top)$, since this would imply an occurrence of $u = 010$, which should be verified by the states from Q_{vfy}, more precisely, by the state sequence $q_0(\text{vfy}, 1, \top\bot)(\text{vfy}, 2, \bot\top)q_0$.

We are ready to present the construction of $\mathcal{B}_{\mathcal{A}_1, u, T_z}$. The NFA $\mathcal{B}_{\mathcal{A}_1, u, T_z}$ is constructed by the following three-step procedure.

(1) Construct the product automaton $\mathcal{A}_1 \times \mathcal{A}_u$. Note that the initial state of $\mathcal{A}_1 \times \mathcal{A}_u$ is (q_0, q_0) and the set of final states of $\mathcal{A}_1 \times \mathcal{A}_u$ is $F_1 \times F_u$.
(2) Remove from $\mathcal{A}_1 \times \mathcal{A}_u$ all the (incoming or outgoing) transitions associated with the states from $Q_1 \times Q_{\text{vfy}}$.
(3) For each pair $(q, q') \in T_z$ and each sequence of transitions in $\mathcal{A}_u$ of the form

$$\left(p, u[1], \left(\text{vfy}, 1, \overrightarrow{W_1'}\right)\right), \left(\left(\text{vfy}, 1, \overrightarrow{W_1'}\right), u[2], \left(\text{vfy}, 2, \overrightarrow{W_2'}\right)\right), \cdots, \left(\left(\text{vfy}, k-1, \overrightarrow{W_{k-1}'}\right), u[k], q_0\right),$$

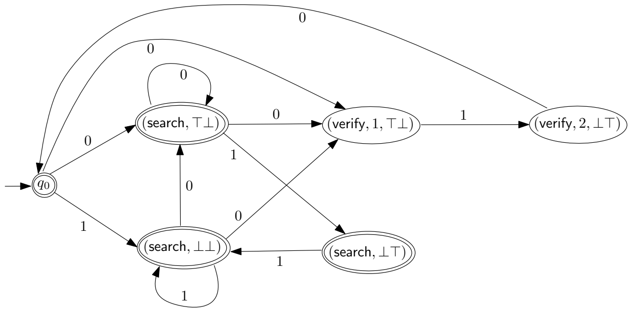


Fig. 5. The parsing automaton $\mathcal{A}_u$ for $u = 010$

where $p = q_0$ or $p = \left(\text{search}, \overrightarrow{W}\right)$, add the following transitions

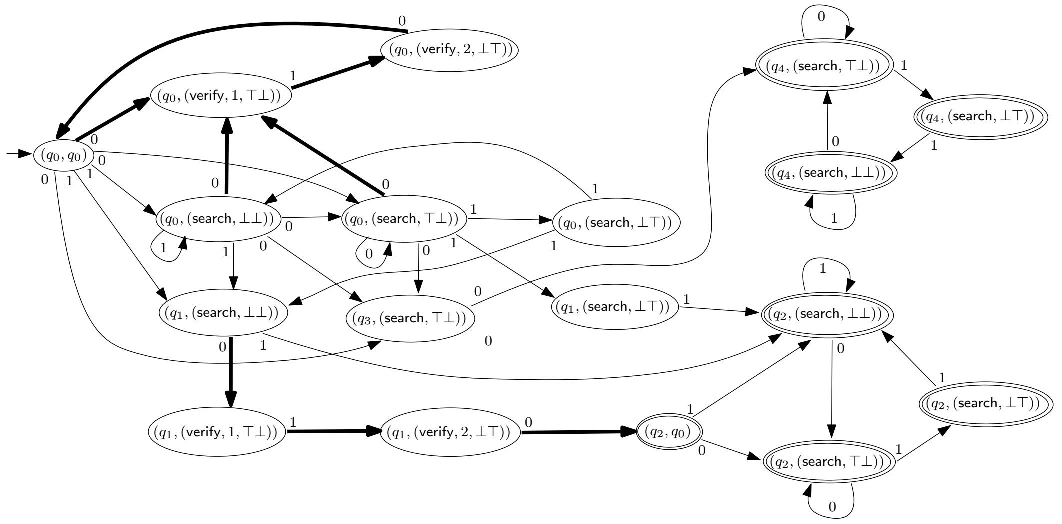
$$\left((q,p), u[1], \left(q, \left(\text{vfy}, 1, \overrightarrow{W_1'}\right)\right)\right), \left(\left(q, \left(\text{vfy}, 1, \overrightarrow{W_1'}\right)\right), u[2], \left(q, \left(\text{vfy}, 2, \overrightarrow{W_2'}\right)\right)\right), \cdots,$$
$$\left(\left(q, \left(\text{vfy}, k-2, \overrightarrow{W_{k-2}'}\right)\right), u[k-1], \left(q, \left(\text{vfy}, k-1, \overrightarrow{W_{k-1}'}\right)\right)\right), \left(\left(q, \left(\text{vfy}, k-1, \overrightarrow{W_{k-1}'}\right)\right), u[k], (q', q_0)\right).$$

Note that the number of aforementioned sequences of transitions in $\mathcal{A}_u$ is at most $|Q_{\text{search}}|+1$, since $\overrightarrow{W_1'}, \ldots, \overrightarrow{W_{k-1}'}$ are completely determined by $\overrightarrow{W}$ and u. Intuitively, when $\mathcal{A}_u$ identifies an occurrence of u, if the current state of $\mathcal{A}_1$ is q, then after reading the occurrence of u, $\mathcal{B}_{\mathcal{A}_1,u,T_z}$ jumps from q to some state q' such that $(q, q') \in T_z$.

Example 7.6. Consider $C \equiv x = \text{replaceAll}(y, u, z) \wedge x \in e_1 \wedge y \in e_2 \wedge z \in e_3$, where $u = 010$, and e_1, e_2, e_3 are as in Example 6.2 (cf. Figure 2). Let $T_z = \{(q_0, q_0), (q_1, q_2)\}$. The NFA $\mathcal{B}_{\mathcal{A}_1,u,T_z}$ is obtained from the product automaton $\mathcal{A}_1 \times \mathcal{A}_u$ (which we give in the full version for reference) by first removing all the transitions associated with the states from $Q_1 \times Q_{\text{vfy}}$, then adding the transitions according to T_z as aforementioned (see Figure 6, where thick edges indicate added transitions). It is routine to check that 01010101 is accepted by $\mathcal{B}_{\mathcal{A}_1,u,T_z}$ and $\mathcal{A}_2$. Moreover, $10 \in \mathcal{L}(\mathcal{A}_3) \cap \mathcal{L}(\mathcal{A}_1(q_0, q_0)) \cap \mathcal{L}(\mathcal{A}_1(q_1, q_2))$. Let y be 01010101 and z be 10. Then x takes the value $\text{replaceAll}(01010101, 010, 10) = 101101$, which is accepted by $\mathcal{A}_1$. Therefore, C is satisfiable.

For the more general case that the SL[replaceAll] formula C contains more than one occurrence of replaceAll$(-,-,-)$ terms, similar to the single-letter case in Section 6, we can nondeterministically remove the edges in the dependency graph G_C in a top-down manner and reduce the satisfiability of C to the satisfiability of a collection of regular constraints for source variables.

Complexity. When constructing G_{i+1} from G_i, suppose the two edges from x to y and z respectively are currently removed, let the labels of the two edges be (l, u) and (r, u) respectively. Then each element $(\mathcal{T}, \mathcal{P})$ of $\mathcal{E}_i(x)$ may be transformed into an element $(\mathcal{T}', \mathcal{P}')$ of $\mathcal{E}_{i+1}(y)$ such that $|\mathcal{T}'| = O(|u||\mathcal{T}|)$, meanwhile, it may also be transformed into an element $(\mathcal{T}'', \mathcal{P}'')$ of $\mathcal{E}_{i+1}(z)$ such that $\mathcal{T}''$ has the same state space as $\mathcal{T}$. In each step of the decision procedure, the state space of the regular constraints may be multiplied by a factor $|u|$. The state space of these regular constraints is at most exponential in the end, so that we can still solve the nonemptiness problem of the intersection of all these regular constraints in exponential space. In addition, if the l-length of G_C is bounded by a constant c, then for each source variable, we get polynomially many regular constraints, where each of them has a state space of polynomial size. Therefore, we can get a polynomial space algorithm. See the full version for a detailed analysis.



(q_0, q_0)
$(q_0, (\text{verify}, 1, \top\bot))$
$(q_0, (\text{verify}, 2, \bot\top))$
$(q_0, (\text{search}, \bot\bot))$
$(q_0, (\text{search}, \top\bot))$
$(q_0, (\text{search}, \bot\top))$
$(q_1, (\text{search}, \bot\bot))$
$(q_3, (\text{search}, \top\bot))$
$(q_1, (\text{search}, \bot\top))$
$(q_1, (\text{verify}, 1, \top\bot))$
$(q_1, (\text{verify}, 2, \bot\top))$
(q_2, q_0)
$(q_2, (\text{search}, \bot\bot))$
$(q_2, (\text{search}, \bot\top))$
$(q_2, (\text{search}, \top\bot))$
$(q_4, (\text{search}, \top\bot))$
$(q_4, (\text{search}, \bot\top))$
$(q_4, (\text{search}, \bot\bot))$

Fig. 6. The NFA $\mathcal{B}_{\mathcal{A}_1, u, T_z}$ for $u = 010$ and $T_z = \{(q_0, q_0), (q_1, q_2)\}$

8 DECISION PROCEDURE FOR SL[replaceAll]: THE REGULAR-EXPRESSION CASE

We consider the case that the second parameter of the replaceAll function is a regular expression. The decision procedure presented below is a generalisation of those in Section 6 and Section 7.

As in the previous sections, we will again start with the simple situation that $C \equiv x = \text{replaceAll}(y, e_0, z) \wedge x \in e_1 \wedge y \in e_2 \wedge z \in e_3$. For $0 \leq i \leq 3$, let $\mathcal{A}_i = (Q_i, \delta_i, q_{0,i}, F_i)$ be the NFA corresponding to e_i.

Let us first consider the special case $\mathcal{L}(e_0) = \{\varepsilon\}$. Then according to the semantics, for each string $u = a_1 \cdots a_n$, $\text{replaceAll}(u, e_0, v) = va_1v \cdots va_nv$. We can solve the satisfiability of C as follows:

(1) Guess a set $T_z \subseteq Q_1 \times Q_1$.
(2) Construct $\mathcal{B}_{\mathcal{A}_1, \varepsilon, T_z}$ from $\mathcal{A}_1$ and T_z as follows: For each $(q, q') \in T_z$, add to $\mathcal{A}_1$ a transition (q, ε, q'). Then transform the resulting NFA into one without ε-transitions (which can be done in polynomial time).
(3) Decide the nonemptiness of $\mathcal{L}(\mathcal{A}_2) \cap \mathcal{L}(\mathcal{B}_{\mathcal{A}_1, \varepsilon, T_z})$ and $\mathcal{L}(\mathcal{A}_3) \cap \bigcap\limits_{(q,q') \in T_z} \mathcal{L}(\mathcal{A}_1(q, q'))$.

Next, let us assume that $\mathcal{L}(e_0) \neq \{\varepsilon\}$. For simplicity of presentation, we assume $\varepsilon \notin \mathcal{L}(e_0)$. The case that $\varepsilon \in \mathcal{L}(e_0)$ can be dealt with in a slightly more technical albeit similar way.

Since $\varepsilon \notin \mathcal{L}(e_0)$, we have $q_{0,0} \notin F_0$. In addition, without loss of generality, we assume that there are no incoming transitions for $q_{0,0}$ in $\mathcal{A}_0$.

To check the satisfiability of C, similar to the constant-string case, we construct a parsing automaton $\mathcal{A}_{e_0}$ that parses a string $v \in \Sigma^* e_0 \Sigma^*$ into $v_1 u_1 v_2 u_2 \ldots v_l u_l v_{l+1}$ such that

- for each $j \in [l]$, u_j is the leftmost and longest matching of e_0 in $(v_1 u_1 \ldots v_{j-1} u_{j-1})^{-1} v$,
- $v_{l+1} \notin \Sigma^* e_0 \Sigma^*$.

We will first give an intuitive description of the behaviour of the automaton $\mathcal{A}_{e_0}$. We start with an automaton that can have an infinite number of states and describe the automaton as starting new "threads", i.e., run multiple copies of $\mathcal{A}_0$ on the input word (similar to alternating automata). We also show how this automaton can be implemented using only a finite number of states. Intuitively, in order to search for the leftmost and longest matching of e_0, $\mathcal{A}_{e_0}$ behaves as follows.

- $\mathcal{A}_{e_0}$ has two modes, "left" and "long", which intuitively means searching for the first and last position of the leftmost and longest matching of e_0 respectively.
- When in the "left" mode, $\mathcal{A}_{e_0}$ starts a new thread of $\mathcal{A}_0$ in each position and records *the set of states* of the threads into a vector. In addition, it nondeterministically makes a "leftmost" guessing, that is, guesses that the current position is the first position of the leftmost and longest matching. If it makes such a guessing, it enters the "long" mode, runs the thread started in the current position and searches for the last position of the leftmost and longest matching. Moreover, it stores in a set S the union of the sets of states of all the threads that were started before the current position and continues running these threads to make sure that, in these threads, the final states will not be reached (thus, the current position is indeed the first position of the leftmost and longest matching).
- When in the "long" mode, $\mathcal{A}_{e_0}$ runs a thread of $\mathcal{A}_0$ to search for the last position of the leftmost and longest matching. If the set of states of the thread contains a final state, then $\mathcal{A}_{e_0}$ nondeterministically guesses that the current position is the last position of the leftmost and longest matching. If it makes such a guessing, then it resets the set of states of the thread and starts a new round of searching for the leftmost and longest matching. In addition, it stores the original set of states of the thread into a set S and continues running the thread to make sure that in this thread, the final states will not be reached (thus, the current position is indeed the last position of the leftmost and longest matching).
- Since the length of the vectors of the sets of states of the threads may become unbounded, in order to obtain a finite state automaton, the following trick is applied. Suppose that the vector is $S_1S_2\cdots S_n$. For each pair of indices $i, j : i < j$ and each $q \in S_i \cap S_j$, remove q from S_j. The application of this trick is justified by the following arguments: Since q occurs in both S_i and S_j and the thread i was started before the thread j, even if from q a final state can be reached in the future, the position where the thread j was started *cannot* be the first position of the leftmost and longest matching, since the state q is also a state of the thread i and the position where the thread i was started is before the position where the thread i was started.

Before presenting the construction of $\mathcal{A}_{e_0}$ in detail, let us introduce some additional notation.

For $S \subseteq Q_0$ and $a \in \Sigma$, let $\delta_0(S, a)$ denote $\{q' \in Q_0 \mid \exists q \in S.\ (q, a, q') \in \delta_0\}$. For $a \in \Sigma$ and a vector $\rho = S_1 \cdots S_n$ such that $S_i \subseteq Q_0$ for each $i \in [n]$, let $\delta_0(\rho, a) = \delta_0(S_1, a) \cdots \delta_0(S_n, a)$.

For a vector $S_1 \cdots S_n$ such that $S_i \subseteq Q_0$ for each $i \in [n]$, we define $\mathsf{red}(S_1 \cdots S_n)$ inductively:

- If $n = 1$, then $\mathsf{red}(S_1) = S_1$ if $S_1 \neq \emptyset$, and $\mathsf{red}(S_1) = \varepsilon$ otherwise.
- If $n > 1$, then

$$\mathsf{red}(S_1 \cdots S_n) = \begin{cases} \mathsf{red}(S_1 \cdots S_{n-1}) & \text{if } S_n \subseteq \bigcup\limits_{i \in [n-1]} S_i, \\ \mathsf{red}(S_1 \cdots S_{n-1})(S_n \setminus \bigcup\limits_{i \in [n-1]} S_i) & \text{o/w} \end{cases}$$

For instance, $\mathsf{red}(\emptyset\{q\}) = \{q\}$ and

$$\mathsf{red}(\{q_1, q_2\}\{q_1, q_3\}\{q_2, q_4\}) = \mathsf{red}(\{q_1, q_2\}\{q_1, q_3\})\{q_4\} = \mathsf{red}(\{q_1, q_2\})\{q_3\}\{q_4\} = \{q_1, q_2\}\{q_3\}\{q_4\}.$$

We give the formal description of $\mathcal{A}_{e_0} = (Q_{e_0}, \delta_{e_0}, q_{0,e_0}, F_{e_0})$ below. The automaton will contain states of the form (ρ, m, S) where ρ is the vector $S_1 \cdots S_n$ recording the set of states of the threads of $\mathcal{A}_0$. The second component m is either left or long indicating the mode. Finally S is the set of states representing all threads for which final states must not be reached.

- Q_{e_0} comprises
  - the tuples $(\{q_{0,0}\}, \mathsf{left}, S)$ such that $S \subseteq Q_0$,

  - the tuples $(\rho\{q_{0,0}\}, \mathsf{left}, S)$ such that $\rho = S_1 \cdots S_n$ with $n \geq 1$ satisfying that for each $i \in [n]$, $S_i \subseteq Q_0 \setminus \{q_{0,0}\}$, and for each pair of indices $i, j : i < j$, $S_i \cap S_j = \emptyset$, moreover, $S \subseteq Q_0 \setminus F_0$,
  - the tuples (S_1, long, S) such that $S_1 \subseteq Q_0$, $S \subseteq Q_0 \setminus F_0$ and $S_1 \not\subseteq S$;
- $q_{0,e_0} = (\{q_{0,0}\}, \mathsf{left}, \emptyset)$,
- F_{e_0} comprises the states of the form $(-, \mathsf{left}, -) \in Q_{e_0}$,
- δ_{e_0} is defined as follows:
  - (continue left) suppose $(\rho\{q_{0,0}\}, \mathsf{left}, S) \in Q_{e_0}$ such that $\rho = S_1 \cdots S_n$ with $n \geq 0$ ($n = 0$ means that ρ is empty), $a \in \Sigma$, $\left(\bigcup_{j\in[n]} \delta_0(S_j, a) \cup \delta_0(\{q_{0,0}\}, a)\right) \cap F_0 = \emptyset$, and $\delta_0(S, a) \cap F_0 = \emptyset$, then

$$\left((\rho\{q_{0,0}\}, \mathsf{left}, S), a, \left(\mathsf{red}(\delta_0(\rho\{q_{0,0}\}, a))\{q_{0,0}\}, \mathsf{left}, \delta_0(S, a)\right)\right) \in \delta_{e_0},$$

Intuitively, in a state (ρ, left, S), if $\left(\bigcup_{j\in[n]} \delta_0(S_j, a) \cup \delta_0(\{q_{0,0}\}, a)\right) \cap F_0 = \emptyset$ and $\delta_0(S, a) \cap F_0 = \emptyset$, then $\mathcal{A}_{e_0}$ can choose to stay in the "left" mode. Moreover, no states occur more than once in $\mathsf{red}(\delta_0(\rho\{q_{0,0}\}, a))\{q_{0,0}\}$, since $q_{0,0}$ does not occur in $\mathsf{red}(\delta_0(\rho\{q_{0,0}\}, a))$, (from the assumption that there are no incoming transitions for $q_{0,0}$ in $\mathcal{A}_0$),
  - (start long) suppose $(\rho\{q_{0,0}\}, \mathsf{left}, S) \in Q_{e_0}$ such that $\rho = S_1 \cdots S_n$ with $n \geq 0$, $a \in \Sigma$, $\delta_0(S, a) \cap F_0 = \emptyset$, $\left(\bigcup_{j\in[n]} \delta_0(S_j, a)\right) \cap F_0 = \emptyset$, and $\delta_0(\{q_{0,0}\}, a) \not\subseteq \delta_0(S, a) \cup \bigcup_{j\in[n]} \delta_0(S_j, a)$, then

$$\left((\rho\{q_{0,0}\}, \mathsf{left}, S), a, \left(\delta_0(\{q_{0,0}\}, a), \mathsf{long}, \delta_0(S, a) \cup \bigcup_{j\in[n]} \delta_0(S_j, a)\right)\right) \in \delta_{e_0}.$$

Intuitively, from a state $(\rho\{q_{0,0}\}, \mathsf{left}, S)$ with $\rho = S_1 \cdots S_n$, when reading a letter a, if $\left(\bigcup_{j\in[n]} \delta_0(S_j, a)\right) \cap F_0 = \emptyset$, $\delta_0(S, a) \cap F_0 = \emptyset$, and $\delta_0(\{q_{0,0}\}, a) \not\subseteq \delta_0(S, a) \cup \bigcup_{j\in[n]} \delta_0(S_j, a)$, then $\mathcal{A}_{e_0}$ guesses that the current position is the first position of the leftmost and longest matching, it goes to the "long" mode, in addition, it keeps in the first component of the control state only the set of states of the thread started in the current position, and puts the union of the sets of the states of all the threads that have been started before, namely, $\bigcup_{j\in[n]} \delta_0(S_j, a)$, into the third component to guarantee that none of these threads will reach a final state in the future (thus the guessing that the current position is the first position of the leftmost and longest matching is correct),
  - (continue long) suppose $(S_1, \mathsf{long}, S) \in Q_{e_0}$, $\delta_0(S, a) \cap F_0 = \emptyset$, and $\delta_0(S_1, a) \not\subseteq \delta_0(S, a)$, then

$$((S_1, \mathsf{long}, S), a, (\delta_0(S_1, a), \mathsf{long}, \delta_0(S, a))) \in \delta_{e_0},$$

intuitively, $\mathcal{A}_{e_0}$ guesses that the current position is not the last position of the leftmost and longest matching and continues the "long" mode,
  - (end long) suppose $(S_1, \mathsf{long}, S) \in Q_{e_0}$, $\delta_0(S_1, a) \cap F_0 \neq \emptyset$, and $\delta_0(S, a) \cap F_0 = \emptyset$, then

$$((S_1, \mathsf{long}, S), a, (\{q_{0,0}\}, \mathsf{left}, \delta_0(S, a) \cup \delta_0(S_1, a))) \in \delta_{e_0},$$

intuitively, when $\delta_0(S_1, a) \cap F_0 \neq \emptyset$ and $\delta_0(S, a) \cap F_0 = \emptyset$, $\mathcal{A}_{e_0}$ guesses that the current position is the last position of the leftmost and longest matching, resets the first component to $\{q_{0,0}\}$, goes to the "left" mode, and puts $\delta_0(S_1, a)$ to the third component to guarantee that the current thread will not reach a final state in the future (thus the guessing that the current position is the last position of the leftmost and longest matching is correct).

– (a matches e_0) suppose $(\rho\{q_{0,0}\}, \mathsf{left}, S) \in Q_{e_0}$ such that $\rho = S_1 \cdots S_n$ with $n \geq 0$, $a \in \Sigma$, $\big(\bigcup\limits_{j \in [n]} \delta_0(S_j, a)\big) \cap F_0 = \emptyset$, $\delta_0(\{q_{0,0}\}, a) \cap F_0 \neq \emptyset$, and $\delta_0(S, a) \cap F_0 = \emptyset$, then

$$\left((\rho\{q_{0,0}\}, \mathsf{left}, S), a, \left(\{q_{0,0}\}, \mathsf{left}, \delta_0(S, a) \cup \bigcup_{j \in [n]} \delta_0(S_j, a) \cup \delta_0(\{q_{0,0}\}, a)\right)\right) \in \delta_{e_0},$$

intuitively, from a state $(\rho\{q_{0,0}\}, \mathsf{left}, S)$ with $\rho = S_1 \cdots S_n$, when reading a letter a, if $\big(\bigcup\limits_{j \in [n]} \delta_0(S_j, a)\big) \cap F_0 = \emptyset$, $\delta_0(\{q_{0,0}\}, a) \cap F_0 \neq \emptyset$, and $\delta_0(S, a) \cap F_0 = \emptyset$, then $\mathcal{A}_{e_0}$ guesses that a is simply the leftmost and longest matching of e_0 (e.g. when $e_0 = a$), then it directly goes to the "left" mode (without going to the "long" mode), resets the first component of the control state to $\{q_{0,0}\}$, and puts the union of the sets of the states of all the threads that have been started, including the one started in the current position, namely, $\bigcup\limits_{j \in [n]} \delta_0(S_j, a) \cup \delta_0(\{q_{0,0}\}, a)$, into the third component to guarantee that none of these threads will reach a final state in the future (where $\bigcup\limits_{j \in [n]} \delta_0(S_j, a)$ is used to validate the leftmost guessing and $\delta_0(\{q_{0,0}\}, a)$ is used to validate the longest guessing).

Let $Q_{\mathsf{left}} = \{(-, \mathsf{left}, -) \in Q_{e_0}\}$, $Q_{\mathsf{long}} = \{(-, \mathsf{long}, -) \in Q_{e_0}\}$, and $v = v_1 u_1 v_2 u_2 \cdots v_l u_l v_{l+1}$ such that u_j is the leftmost and longest matching of e_0 in $(v_1 u_1 \cdots v_{j-1} u_{j-1})^{-1} v$ for each $j \in [l]$, in addition, $v_{l+1} \notin \Sigma^* e \Sigma^*$. Then there exists a *unique* accepting run r of $\mathcal{A}_{e_0}$ on v such that the state sequence in r is of the form

$$(\{q_{0,0}\}, \mathsf{left}, \emptyset)\ r_1\ (\{q_{0,0}\}, \mathsf{left}, -)\ r_2\ (\{q_{0,0}\}, \mathsf{left}, -) \cdots r_l\ (\{q_{0,0}\}, \mathsf{left}, -)\ r_{l+1},$$

where for each $j \in [l]$, $r_j \in \mathcal{L}((Q_{\mathsf{left}})^* \circ (Q_{\mathsf{long}})^*)$, and $r_{l+1} \in \mathcal{L}((Q_{\mathsf{left}})^*)$. Intuitively, each occurrence of the state subsequence from $\mathcal{L}((Q_{\mathsf{long}})^* \circ (\{q_{0,0}\}, \mathsf{left}, -))$, except the first one, witnesses the *leftmost and longest* matching of e_0 in v from the beginning or after the previous such a matching.

Since in the first component $\rho q_{0,0}$ of each state of $\mathcal{A}_{e_0}$, no states from $\mathcal{A}_0$ occur more than once, it is not hard to see that $|\mathcal{A}_{e_0}|$ is $2^{O(p(|\mathcal{A}_0|))}$ for some polynomial p.

Given $T_z \subseteq Q_1 \times Q_1$, we construct $\mathcal{B}_{\mathcal{A}_1, e_0, T_z}$ by the following three-step procedure.

(1) Construct the product of $\mathcal{A}_1$ and $\mathcal{A}_{e_0}$.
(2) Remove all transitions associated with states from $Q_1 \times Q_{\mathsf{long}}$, in addition, remove all transitions of the form $((q, (\rho\{q_{0,0}\}, \mathsf{left}, S)), a, (q', (\{q_{0,0}\}, \mathsf{left}, S')))$ such that $\delta_0(q_{0,0}, a) \cap F_0 \neq \emptyset$.
(3) For each pair $(q, q') \in T_z$, do the following,
   - for each transition

$$\left((\rho\{q_{0,0}\}, \mathsf{left}, S), a, \left(\delta_0(\{q_{0,0}\}, a), \mathsf{long}, \delta_0(S, a) \cup \bigcup_{j \in [n]} \delta_0(S_j, a)\right)\right) \in \delta_{e_0},$$

   add a transition $\left(\left(q, (\rho\{q_{0,0}\}, \mathsf{left}, S)\right), a, \left(q, \left(\delta_0(\{q_{0,0}\}, a), \mathsf{long}, \delta_0(S, a) \cup \bigcup\limits_{j \in [n]} \delta_0(S_j, a)\right)\right)\right)$,
   - for each transition

$$((S_1, \mathsf{long}, S), a, (\delta_0(S_1, a), \mathsf{long}, \delta_0(S, a))) \in \delta_{e_0},$$

   add a transition $((q, (S_1, \mathsf{long}, S)), a, (q, (\delta_0(S_1, a), \mathsf{long}, \delta_0(S, a))))$,
   - for each transition

$$((S_1, \mathsf{long}, S), a, (\{q_{0,0}\}, \mathsf{left}, \delta_0(S, a) \cup \delta_0(S_1, a))) \in \delta_{e_0},$$

   add a transition $((q, (S_1, \mathsf{long}, S)), a, (q', (\{q_{0,0}\}, \mathsf{left}, \delta_0(S, a) \cup \delta_0(S_1, a))))$,

- for each $\left((\rho\{q_{0,0}\}, \mathsf{left}, S), a, \left(\{q_{0,0}\}, \mathsf{left}, \delta_0(S, a) \cup \bigcup_{j \in [n]} \delta_0(S_j, a) \cup \delta_0(\{q_{0,0}\}, a)\right)\right) \in \delta_{e_0}$, add a transition
$$\left((q, (\rho\{q_{0,0}\}, \mathsf{left}, S)), a, \left(q', \left(\{q_{0,0}\}, \mathsf{left}, \delta_0(S, a) \cup \bigcup_{j \in [n]} \delta_0(S_j, a) \cup \delta_0(\{q_{0,0}\}, a)\right)\right)\right).$$

Since $|\mathcal{A}_{e_0}|$ is $2^{O(p(|\mathcal{A}_0|))}$, it follows that $|\mathcal{B}_{\mathcal{A}_1, e_0, T_z}|$ is $|\mathcal{A}_1| \cdot 2^{O(p(|\mathcal{A}_0|))}$. In addition, since $|\mathcal{A}_0| = O(|e_0|)$, we deduce that $|\mathcal{B}_{\mathcal{A}_1, e_0, T_z}|$ is $|\mathcal{A}_1| \cdot 2^{O(p(|e_0|))}$.

For the more general case that the SL[replaceAll] formula C contains more than one occurrence of replaceAll$(-, -, -)$ terms, we still nondeterministically remove the edges in the dependency graph G_C in a top-down manner and reduce the satisfiability of C to the satisfiability of a collection of regular constraints for source variables.

Complexity. In each step of the reduction, suppose the two edges out of x are currently removed, let the two edges be from x to y and z and labeled by (l, e) and (r, e) respectively, then each element of $(\mathcal{T}, \mathcal{P})$ of $\mathcal{E}_i(x)$ may be transformed into an element $(\mathcal{T}', \mathcal{P}')$ of $\mathcal{E}_{i+1}(y)$ such that $|\mathcal{T}'| = |\mathcal{T}| \cdot 2^{O(p(|e|))}$, meanwhile, it may also be transformed into an element $(\mathcal{T}'', \mathcal{P}'')$ of $\mathcal{E}_{i+1}(y)$ such that $\mathcal{T}''$ has the same state space as $\mathcal{T}$. Thus, after the reduction, for each source variable x, $\mathcal{E}(x)$ may contain exponentially many elements, and each of them may have a state space of exponential size. To solve the nonemptiness problem of the intersection of all these regular constraints, the exponential space is sufficient. In addition, if the l-length of G_C is at most one, we can show that for each source variable x, $\mathcal{E}(x)$ corresponds to the intersection of polynomially many regular constraints, where each of them has a state space at most exponential size. To solve the nonemptiness of the intersection of these regular constraints, a polynomial space is sufficient. See the full version for a detailed analysis.

9 UNDECIDABLE EXTENSIONS

In this section, we consider the language SL[replaceAll] extended with either integer constraints, character constraints, or IndexOf constraints, and show that each of such extensions leads to undecidability. We will use variables of, in additional to the type Str, the Integer data type Int. The type Str consists of the string variables as in the previous sections. A variable of type Int, usually referred to as an *integer variable*, ranges over the set $\mathbb{N}$ of natural numbers. Recall that, in previous sections, we have used $x, y, z, \ldots$ to denote the variables of Str type. Hereafter we typically use $\mathfrak{l}, \mathfrak{m}, \mathfrak{n}, \ldots$ to denote the variables of Int. The choice of omitting negative integers is for simplicity. Our results can be easily extended to the case where Int includes negative integers.

We begin by defining the kinds of constraints we will use to extend SL[replaceAll]. First, we describe integer constraints, which express constraints on the length or number of occurrences of symbols in words.

Definition 9.1 (Integer constraints). An atomic integer constraint over Σ is an expression of the form $a_1 t_1 + \cdots + a_n t_n \leq d$ where $a_1, \cdots, a_n, d \in \mathbb{Z}$ are constant integers (represented in binary), and each *term* t_i is either

(1) an integer variable $\mathfrak{n}$;
(2) $|x|$ where x is a string variable; or
(3) $|x|_a$ where x is string variable and $a \in \Sigma$ is a constant letter.

Here, $|x|$ and $|x|_a$ denote the length of x and the number of occurrences of a in x, respectively.

An *integer constraint* over Σ is a Boolean combination of atomic integer constraints over Σ.

Character constraints, on the other hand, allow to compare symbols from different strings. The formal definitions are given as follows.

Definition 9.2 (Character constraints). An *atomic character constraint* over Σ is an equation of the form $x[t_1] = y[t_2]$ where

- x and y are either a string variable or a constant string in Σ^*, and
- t_1 and t_2 are either integer variables or constant positive integers.

Here, the interpretation of $x[t_1]$ is the t_1-th letter of x. In case that x does not have the t_1-th letter *or* y does not have the t_2-th letter, the constraint $x[t_1] = y[t_2]$ is false by convention.

A *character constraint* over Σ is a Boolean combination of atomic character constraints over Σ.

We also consider the constraints involving the IndexOf function.

*Definition 9.3 (*IndexOf *Constraints).* An atomic IndexOf constraint over Σ is a formula of the form $t \mathfrak{o}$ IndexOf(s_1, s_2), where

- t is an integer variable, or a positive integer (recall that here we assume that the first position of a string is 1), or the value 0 (denoting that there is no occurrence of s_1 in s_2),
- $\mathfrak{o} \in \{\geq, \leq\}$, and
- s_1, s_2 are either string variables or constant strings.

We consider the *first-occurrence* semantics of IndexOf. More specifically, $t \geq$ IndexOf(s_1, s_2) holds if t is no less than the first position in s_2 where s_1 occurs, similarly for $t \leq$ IndexOf(s_1, s_2).

An IndexOf constraint over Σ is a Boolean combination of atomic IndexOf constraints over Σ.

We will show that the extension of SL[replaceAll] with integer constraints entails undecidability, by a reduction from (a variant of) the Hilbert's 10th problem, which is well-known to be undecidable [Matiyasevich 1993]. For space reasons, all proofs appear in the full version. Intuitively, we want to find a solution to $f(x_1, \cdots, x_n) = g(x_1, \cdots, x_n)$ in the natural numbers, where f and g are polynomials with positive coefficients. We can use the length of string variables over a unary alphabet $\{a\}$ to represent integer variables, addition can be performed with concatenation, and multiplication of x and y with replaceAll(x, a, y). The integer constraint $|x| = |y|$ asserts the equality of f and g. Note that the use of concatenation can be further dispensed since, by Proposition 3.6, concatenation is expressible by replaceAll at the price of a slightly extended alphabet.

Theorem 9.4. *For the extension of* SL[replaceAll] *with* integer constraints, *the satisfiability problem is undecidable, even if only a single integer constraint of the form $|x| = |y|$ or $|x|_a = |y|_a$ is used.*

Notice that the extension of SL[replaceAll] with only one integer constraint of the form $|x| = |y|$ entails undecidability. We remark that the undecidability result here does *not* follow from the undecidability result for the extension of word equations with the letter-counting modalities in [Büchi and Senger 1990], since the formula by [Büchi and Senger 1990] is not straight-line.

By utilising a further result on Diophantine equations, we show that for the extension of SL[replaceAll] with integer constraints, even if the SL[replaceAll] formulae are simple (in the sense that their dependency graphs are of depth at most one), the satisfiability problem is still undecidable (note that no restrictions are put on the integer constraints in this case).

Theorem 9.5. *For the extension of* SL[replaceAll] *with integer constraints, even if* SL[replaceAll] *formulae are restricted to those whose dependency graphs are of depth at most one, the satisfiability problem is still undecidable.*

By essentially encoding $|x| = |y|$ with *character* or IndexOf *constraints*, we show:

Proposition 9.6. *For the extension of* SL[replaceAll] *with either the character constraints or the* IndexOf *constraints, the satisfiability problem is undecidable.*

10 RELATED WORK

We now discuss some related work. We split our discussion into two categories: (1) theoretical results in terms of decidability and complexity; (2) practical (but generally incomplete) approaches used in string solvers. We emphasise work on replaceAll functions as they are our focus.

Theoretical Results. We have discussed in Section 1 works on string constraints with the theory of strings with concatenation. This research programme builds on the question of solving satisfiability of *word equations*, i.e., a string equation $\alpha = \beta$ containing concatenation of string constants and variables. Makanin showed decidability [Makanin 1977], whose upper bound was improved to PSPACE in [Plandowski 2004] using a word compression technique. A simpler algorithm was in recent years proposed in [Jez 2017] using the recompression technique. The best lower bound for this problem is still NP, and closing this complexity gap is a long-standing open problem. Decidability (in fact, the PSPACE upper bound) can be retained in the presence of regular constraints (e.g. see [Schulz 1990]). This can be extended to existential theory of concatenation with regular constraints using the technique of [Büchi and Senger 1990]. The replace-all operator cannot be expressed by the concatenation operator alone. For this reason, our decidability of the fragment of SL[replaceAll] cannot be derived from the results from the theory of concatenation alone.

Regarding the extension with length constraints, it is still a long-standing open problem whether word equations with length constraints is decidable, though it is known that letter-counting (e.g. counting the number of occurrences of 0s and 1s separately) yields undecidability [Büchi and Senger 1990]. It was shown in [Lin and Barceló 2016] that the length constraints (in fact, letter-counting) can be added to the subclass of SL[replaceAll] where the pattern/replacement are constants, while preserving decidability. In contrast, if we allow variables on the replacement parameters of formulas in SL[replaceAll], we can easily encode the Hilbert's 10th problem with length (integer) constraints.

The replaceAll function can be seen as a special, yet expressive, string transformation function, aka string transducer. From this viewpoint, the closest work is [Lin and Barceló 2016], which we discuss extensively in the introduction. Here, we discuss two further recent transducer models: streaming string transducers [Alur and Cerný 2010] and symbolic transducers [Veanes et al. 2012].

A streaming string transducer is a finite state machine where a finite set of string variables are used to store the intermediate results for output. The replaceAll(x, e, y) term can be modelled by an extension of streaming string transducers *with parameters*, that is, a streaming string transducer which reads an input string (interpreted as the value of x), uses y as a free string variable which is presumed to be read-only, and updates a string variable z, which stores the computation result, by a string term which may involve y. Nevertheless, to the best of our knowledge, this extension of streaming string transducers has not been investigated so far.

Symbolic transducers are an extension of Mealy machine to infinite alphabets by using a variable cur to represent the symbol in the current position, and replacing the input and output letters in transitions with unary predicates $\varphi(cur)$ and terms involving cur respectively. Symbolic transducers can model replaceAll functions *when the third parameter is a constant.* Inspired by symbolic transducers, it is perhaps an interesting future work to consider an extension of the replaceAll function by allowing predicates as patterns. For instance, one may consider the term replaceAll$(x, cur \equiv 0 \bmod 2, y)$ which replaces every even number in x with y.

Finally, the replaceAll function is related to Array Folds Logic introduced by Daca et al [Daca et al. 2016]. The authors considered an extension of the quantifier-free theory of integer arrays with counting. The main feature of the logic is the *fold* terms, borrowed from the folding concept in functional programming languages. Intuitively, a fold term applies a function to every element of the array to compute an output. If strings are treated as arrays over a finite domain (the alphabet), the replaceAll function can be seen as a fold term. Nevertheless, the replaceAll function goes beyond the fold terms considered in [Daca et al. 2016], since it outputs a string (an array), instead of an integer. Therefore, the results in [Daca et al. 2016] cannot be applied to our setting.

Practical Solvers. A large amount of recent work develops practical string solvers including Kaluza [Saxena et al. 2010], Hampi [Kiezun et al. 2012], Z3-str [Zheng et al. 2013], CVC4 [Liang et al. 2014], Stranger [Yu et al. 2014], Norn [Abdulla et al. 2014], S3 and S3P [Trinh et al. 2014, 2016], and FAT [Abdulla et al. 2017]. Among them, only Stranger, S3, and S3P support replaceAll.

In the Stranger tool, an automata-based approach was provided for symbolic analysis of PHP programs, where two different semantics replaceAll were considered, namely, the leftmost and longest matching as well as the leftmost and shortest matching. Nevertheless, they focused on the abstract-interpretation based analysis of PHP programs and provided an *over-approximation* of all the possible values of the string variables at each program point. Therefore, their string constraint solving algorithm is *not* an exact decision procedure. In contrast, we provided a decision procedure for the straight-line fragment with the rather general replaceAll function, where the pattern parameter can be arbitrary regular expressions and the replacement parameter can be variables. In the latter case, we consider the leftmost and longest semantics mainly for simplicity, and the decision procedure can be adapted to the leftmost and shortest semantics easily.

The S3 and S3P tools also support the replaceAll function, where some progressive searching strategies were provided to deal with the non-termination problem caused by the recursively defined string operations (of which replaceAll is a special case). Nevertheless, the solvers are incomplete as reasoning about unbounded strings defined recursively is in general an undecidable problem.

11 CONCLUSION

We have initiated a systematic investigation of the decidability of the satisfiability problem for the straight-line fragments of string constraints involving the replaceAll function and regular constraints. The straight-line restriction is known to be appropriate for applications in symbolic execution of string-manipulating programs [Lin and Barceló 2016]. Our main result is a decision procedure for a large fragment of the logic, wherein the pattern parameters are regular expressions (which covers a large proportion of the usage of the replaceAll function in practice). Concatenation is obtained for free since concatenation can be easily expressed in this fragment. We have shown that the decidability of this fragment cannot be substantially extended. This is achieved by showing that if either (1) the pattern parameters are allowed to be variables, or (2) the length constraints are incorporated in the fragment, then we get the undecidability. Our work clarified important fundamental issues surrounding the replaceAll functions in string constraint solving and provided a novel decision procedure which paved a way to a string solver that is able to fully support the replaceAll function. This would be the most immediate future work.

ACKNOWLEDGMENTS

T. Chen is supported by the Australian Research Council under Grant No. DP160101652 and the Engineering and Physical Sciences Research Council under Grant No. EP/P00430X/1. M. Hague is supported by the Engineering and Physical Sciences Research Council under Grant No. EP/K009907/1. A. Lin is supported by the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement no 759969). Z. Wu is supported by the National Natural Science Foundation of China under Grant No. 61472474 and Grant No. 61572478, the INRIA-CAS joint research project "Verification, Interaction, and Proofs",

REFERENCES

Parosh Aziz Abdulla, Mohamed Faouzi Atig, Yu-Fang Chen, Bui Phi Diep, Lukás Holík, Ahmed Rezine, and Philipp Rümmer. 2017. Flatten and conquer: a framework for efficient analysis of string constraints. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017.* 602–617. https://doi.org/10.1145/3062341.3062384

Parosh Aziz Abdulla, Mohamed Faouzi Atig, Yu-Fang Chen, Lukás Holík, Ahmed Rezine, Philipp Rümmer, and Jari Stenman. 2014. String Constraints for Verification. In *CAV*. 150–166.

Rajeev Alur and Pavol Cerný. 2010. Expressiveness of streaming string transducers. In *IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science, FSTTCS 2010, December 15-18, 2010, Chennai, India.* 1–12.

Christel Baier and Joost-Pieter Katoen. 2008. *Principles of Model Checking (Representation and Mind Series).* The MIT Press.

Davide Balzarotti, Marco Cova, Viktoria Felmetsger, Nenad Jovanovic, Engin Kirda, Christopher Kruegel, and Giovanni Vigna. 2008. Saner: Composing Static and Dynamic Analysis to Validate Sanitization in Web Applications. In *2008 IEEE Symposium on Security and Privacy (S&P 2008), 18-21 May 2008, Oakland, California, USA*. 387–401. https://doi.org/10.1109/SP.2008.22

Nikolaj Bjørner, Nikolai Tillmann, and Andrei Voronkov. 2009. Path feasibility analysis for string-manipulating programs. In *TACAS*. 307–321.

J Richard Büchi and Steven Senger. 1990. Definability in the existential theory of concatenation and undecidable extensions of this theory. In *The Collected Works of J. Richard Büchi*. Springer, 671–683.

Cristian Cadar, Vijay Ganesh, Peter M. Pawlowski, David L. Dill, and Dawson R. Engler. 2006. EXE: Automatically Generating Inputs of Death. In *Proceedings of the 13th ACM Conference on Computer and Communications Security (CCS '06)*. ACM, New York, NY, USA, 322–335. https://doi.org/10.1145/1180405.1180445

Przemyslaw Daca, Thomas A. Henzinger, and Andrey Kupriyanov. 2016. Array Folds Logic. In *Computer Aided Verification - 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part II*. 230–248.

Loris D'Antoni and Margus Veanes. 2013. Static Analysis of String Encoders and Decoders. In *VMCAI*. 209–228.

Vijay Ganesh, Mia Minnes, Armando Solar-Lezama, and Martin C. Rinard. 2012. Word Equations with Length Constraints: What's Decidable?. In *Hardware and Software: Verification and Testing - 8th International Haifa Verification Conference, HVC 2012, Haifa, Israel, November 6-8, 2012. Revised Selected Papers*. 209–226. https://doi.org/10.1007/978-3-642-39611-3_21

Patrice Godefroid, Nils Klarlund, and Koushik Sen. 2005. DART: Directed Automated Random Testing. *SIGPLAN Not.* 40, 6 (June 2005), 213–223. https://doi.org/10.1145/1064978.1065036

Google. 2015. Closure Templates. https://developers.google.com/closure/templates/. Referred July 2017.

Pieter Hooimeijer, Benjamin Livshits, David Molnar, Prateek Saxena, and Margus Veanes. 2011. Fast and Precise Sanitizer Analysis with BEK. In *USENIX Security Symposium*.

John E. Hopcroft and Jeffrey D. Ullman. 1979. *Introduction to Automata Theory, Languages and Computation.* Addison-Wesley.

Artur Jez. 2017. Word equations in linear space. *CoRR* abs/1702.00736 (2017). http://arxiv.org/abs/1702.00736

Christoph Kern. 2014. Securing the tangled web. *Commun. ACM* 57, 9 (2014), 38–47. https://doi.org/10.1145/2643134

Adam Kiezun et al. 2012. HAMPI: A solver for word equations over strings, regular expressions, and context-free grammars. *ACM Trans. Softw. Eng. Methodol.* 21, 4 (2012), 25.

James C. King. 1976. Symbolic Execution and Program Testing. *Commun. ACM* 19, 7 (1976), 385–394. https://doi.org/10.1145/360248.360252

Jan Lehnardt and contributors. 2015. mustache.js. https://github.com/janl/mustache.js/. Referred July 2017.

Tianyi Liang, Andrew Reynolds, Cesare Tinelli, Clark Barrett, and Morgan Deters. 2014. A DPLL(T) Theory Solver for a Theory of Strings and Regular Expressions. In *CAV*. 646–662.

Anthony W. Lin and Pablo Barceló. 2016. String Solving with Word Equations and Transducers: Towards a Logic for Analysing Mutation XSS. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '16)*. Springer, 123–136.

Gennady S Makanin. 1977. The problem of solvability of equations in a free semigroup. *Sbornik: Mathematics* 32, 2 (1977), 129–198.

Yuri V. Matiyasevich. 1993. *Hilbert's Tenth Problem.* MIT Press, Cambridge, MA, USA.

K. L. McMillan. 1993. *Symbolic model checking.* Kluwer.

Wojciech Plandowski. 2004. Satisfiability of word equations with constants is in PSPACE. *J. ACM* 51, 3 (2004), 483–496. https://doi.org/10.1145/990308.990312

Prateek Saxena, Devdatta Akhawe, Steve Hanna, Feng Mao, Stephen McCamant, and Dawn Song. 2010. A Symbolic Execution Framework for JavaScript. In *31st IEEE Symposium on Security and Privacy, S&P 2010, 16-19 May 2010, Berleley/Oakland, California, USA*. 513–528. https://doi.org/10.1109/SP.2010.38

Klaus U. Schulz. 1990. Makanin's Algorithm for Word Equations - Two Improvements and a Generalization. In *Word Equations and Related Topics, First International Workshop, IWWERT '90, Tübingen, Germany, October 1-3, 1990, Proceedings*. 85–150. https://doi.org/10.1007/3-540-55124-7_4

Koushik Sen, Swaroop Kalasapur, Tasneem G. Brutch, and Simon Gibbs. 2013. Jalangi: a selective record-replay and dynamic analysis framework for JavaScript. In *Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, ESEC/FSE'13, Saint Petersburg, Russian Federation, August 18-26, 2013*. 488–498. https://doi.org/10.1145/2491411.2491447

Minh-Thai Trinh, Duc-Hiep Chu, and Joxan Jaffar. 2014. S3: A Symbolic String Solver for Vulnerability Detection in Web Applications. In *CCS*. 1232–1243.

Minh-Thai Trinh, Duc-Hiep Chu, and Joxan Jaffar. 2016. Progressive Reasoning over Recursively-Defined Strings. In *Computer Aided Verification - 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part I*. Springer, 218–240.

Margus Veanes, Pieter Hooimeijer, Benjamin Livshits, David Molnar, and Nikolaj Bjørner. 2012. Symbolic finite state transducers: algorithms and applications. In *POPL*. 137–150.

Hung-En Wang, Tzung-Lin Tsai, Chun-Han Lin, Fang Yu, and Jie-Hong R. Jiang. 2016. String Analysis via Automata Manipulation with Logic Circuit Representation. In *Computer Aided Verification - 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part I (Lecture Notes in Computer Science)*, Vol. 9779. Springer, 241–260. https://doi.org/10.1007/978-3-319-41528-4

Chris Wanstrath. 2009. Mustache: Logic-less Templates. https://mustache.github.io/. Referred July 2017.

Jeff Williams, Jim Manico, and Neil Mattatall. 2017. XSS Prevention Cheat Sheet. https://www.owasp.org/index.php/XSS_(Cross_Site_Scripting)_Prevention_Cheat_Sheet. Referred July 2017.

Fang Yu, Muath Alkhalaf, Tevfik Bultan, and Oscar H. Ibarra. 2014. Automata-based Symbolic String Analysis for Vulnerability Detection. *Form. Methods Syst. Des.* 44, 1 (2014), 44–70.

Yunhui Zheng, Xiangyu Zhang, and Vijay Ganesh. 2013. Z3-str: a Z3-based string solver for web application analysis. In *ESEC/SIGSOFT FSE*. 114–124.