Decision Procedures for Path Feasibility of String-Manipulating Programs with Complex Operations

TAOLUE CHEN, Birkbeck, University of London, United Kingdom
MATTHEW HAGUE, Royal Holloway, University of London, United Kingdom
ANTHONY W. LIN, University of Oxford, United Kingdom
PHILIPP RÜMMER, Uppsala University, Sweden
ZHILIN WU, Institute of Software, Chinese Academy of Sciences, China

The design and implementation of decision procedures for checking path feasibility in string-manipulating programs is an important problem, with such applications as symbolic execution of programs with strings and automated detection of cross-site scripting (XSS) vulnerabilities in web applications. A (symbolic) path is given as a finite sequence of assignments and assertions (i.e. without loops), and checking its feasibility amounts to determining the existence of inputs that yield a successful execution. Modern programming languages (e.g. JavaScript, PHP, and Python) support many complex string operations, and strings are also often implicitly modified during a computation in some intricate fashion (e.g. by some autoescaping mechanisms).

In this paper we provide two general semantic conditions which together ensure the decidability of path feasibility: (1) each assertion admits regular monadic decomposition (i.e. is an effectively recognisable relation), and (2) each assignment uses a (possibly nondeterministic) function whose inverse relation preserves regularity. We show that the semantic conditions are *expressive* since they are satisfied by a multitude of string operations including concatenation, one-way and two-way finite-state transducers, replaceAll functions (where the replacement string could contain variables), string-reverse functions, regular-expression matching, and some (restricted) forms of letter-counting/length functions. The semantic conditions also strictly subsume existing decidable string theories (e.g. straight-line fragments, and acyclic logics), and most existing benchmarks (e.g. most of Kaluza's, and all of SLOG's, Stranger's, and SLOTH's benchmarks). Our semantic conditions also yield a conceptually *simple* decision procedure, as well as an *extensible* architecture of a string solver in that a user may easily incorporate his/her own string functions into the solver by simply providing code for the pre-image computation without worrying about other parts of the solver. Despite these, the semantic conditions are unfortunately too general to provide a fast and complete decision procedure. We provide strong theoretical evidence for this in the form of complexity results. To rectify this problem, we propose two solutions. Our main solution is to allow only partial string functions (i.e., prohibit nondeterminism) in condition (2). This restriction is satisfied in many cases in practice, and yields decision procedures that are effective in both theory and practice. Whenever nondeterministic functions are still needed (e.g. the string function split), our second solution is to provide a syntactic fragment that provides a support of nondeterministic functions, and operations like one-way transducers, replaceAll (with constant replacement string), the string-reverse

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; 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; Philipp Rümmer, Department of Information Technology, Uppsala University, Box 337, Uppsala, SE-751 05, Sweden, philipp.ruemmer@it.uu.se; Zhilin Wu, State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences, China.

2475-1421/2019/1-ART49
https://doi.org/10.1145/3290362

function, concatenation, and regular-expression matching. We show that this fragment can be reduced to an existing solver SLOTH that exploits fast model checking algorithms like IC3.

We provide an efficient implementation of our decision procedure (assuming our first solution above, i.e., deterministic partial string functions) in a new string solver OSTRICH. Our implementation provides built-in support for concatenation, reverse, functional transducers (FFT), and replaceAll and provides a framework for extensibility to support further string functions. We demonstrate the efficacy of our new solver against other competitive solvers.

CCS Concepts: • **Theory of computation → Automated reasoning**; **Program verification**; **Regular languages**; *Logic and verification*; Complexity classes;

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

ACM Reference Format:
Taolue Chen, Matthew Hague, Anthony W. Lin, Philipp Rümmer, and Zhilin Wu. 2019. Decision Procedures for Path Feasibility of String-Manipulating Programs with Complex Operations . *Proc. ACM Program. Lang.* 3, POPL, Article 49 (January 2019), 30 pages. https://doi.org/10.1145/3290362

1 INTRODUCTION

Strings are a fundamental data type in virtually all programming languages. Their generic nature can, however, lead to many subtle programming bugs, some with security consequences, e.g., cross-site scripting (XSS), which is among the OWASP Top 10 Application Security Risks [van der Stock et al. 2017]. One effective automatic testing method for identifying subtle programming errors is based on *symbolic execution* [King 1976] and combinations with dynamic analysis called *dynamic symbolic execution* [Cadar et al. 2008, 2006; Godefroid et al. 2005; Sen et al. 2013, 2005]. See [Cadar and Sen 2013] for an excellent survey. Unlike purely random testing, which runs only *concrete* program executions on different inputs, the techniques of symbolic execution analyse *static* paths (also called symbolic executions) through the software system under test. Such a path can be viewed as a constraint φ (over appropriate data domains) and the hope is that a fast solver is available for checking the satisfiability of φ (i.e. to check the *feasibility* of the static path), which can be used for generating inputs that lead to certain parts of the program or an erroneous behaviour.

Constraints from symbolic execution on string-manipulating programs can be understood in terms of the problem of path feasibility over a bounded program S with neither loops nor branching (e.g. see [Bjørner et al. 2009]). That is, S is a sequence of assignments and conditionals/assertions, i.e., generated by the grammar

$$S ::= \qquad y := f(x_1, \ldots, x_r) \mid \mathbf{assert}(g(x_1, \ldots, x_r)) \mid S; S \tag{1}$$

where $f : (\Sigma^*)^r \to \Sigma^*$ is a partial string function and $g \subseteq (\Sigma^*)^r$ is a string relation. The following is a simple example of a symbolic execution S which uses string variables (x, y, and z's) and string constants (letters a and b), and the concatenation operator ($\circ$):

$$z_1 := x \circ \mathsf{ba} \circ y; \quad z_2 := y \circ \mathsf{ab} \circ x; \quad \mathbf{assert}(z_1 == z_2) \tag{2}$$

The problem of *path feasibility/satisfiability*[1] asks whether, for a given program S, there exist *input* strings (e.g. x and y in (2)) that can successfully take S to the end of the program while satisfying all the assertions. This path can be satisfied by assigning y (resp. x) to b (resp. the empty string). In this paper, we will also allow nondeterministic functions $f : (\Sigma^*)^r \to 2^{\Sigma^*}$ since nondeterminism

[1] It is equivalent to satisfiability of string constraints in the SMT framework [Barrett et al. 2009; De Moura and Bjørner 2011; Kroening and Strichman 2008]. Simply convert a symbolic execution S into a *Static Single Assignment* (SSA) form (i.e. use a new variable on l.h.s. of each assignment) and treat assignments as equality, e.g., formula for the above example is $z_1 = x + \mathsf{ba} + y \wedge z_2 = y + \mathsf{ab} + x \ \wedge z_1 = z_2$, where + denotes the string concatenation operation.

can be a useful modelling construct. For example, consider the code in Figure 1. It ensures that each element in `s1` (construed as a list delimited by `-`) is longer than each element in `s2`. If $f : \Sigma^* \to 2^{\Sigma^*}$ is a function that nondeterministically outputs a substring delimited by `-`, our symbolic execution analysis can be reduced to feasibility of the path:

$$x := f(s_1); \quad y := f(s_2); \quad \textbf{assert}(\texttt{len}(x) \leq \texttt{len}(y))$$

In the last few decades much research on the satisfiability problem of string constraints suggests that it takes very little for a string constraint language to become undecidable. For example, although the existential theory of concatenation and regular constraints (i.e. an atomic expression is either $E = E'$, where E and E' are concatenations of string constants and variables, or $x \in L$, where L is a regular language) is decidable and in fact PSPACE-complete [Diekert 2002; Jez 2016; Plandowski 2004], the theory becomes undecidable when enriched with letter-counting [Büchi and Senger 1990], i.e., expressions of the form $|x|_a = |y|_b$, where $|\cdot|_a$ is a function mapping a word to the number of occurrences of the the letter a in the word. Similarly, although finite-state transductions [D'Antoni and Veanes 2013; Hooimeijer et al. 2011; Lin and Barceló 2016] are crucial for expressing many functions used in string-manipulating programs — including autoescaping mechanisms (e.g. backslash escape, and HTML escape in JavaScript), and the replaceAll function with a constant replacement pattern — checking a simple formula of the form $\exists x R(x, x)$, for a given rational transduction[2] R, can easily encode the Post Correspondence Problem [Morvan 2000], and therefore is undecidable.

```
# s1, s2: strings with delimiter '-'
for x in s1.split('-')
  for y in s2.split('-')
    assert(len(x) > len(y))
```

Fig. 1. A Python code snippet

Despite the undecidability of allowing various operations in string constraints, in practice it is common for a string-manipulating program to contain multiple operations (e.g. concatenation and finite-state transductions), and so a path feasibility solver nonetheless needs to be able to handle them. This is one reason why some string solving practitioners opted to support more string operations and settle with incomplete solvers (e.g. with no guarantee of termination) that could still solve some constraints that arise in practice, e.g., see [Abdulla et al. 2017, 2018; Berzish et al. 2017; Kiezun et al. 2012; Liang et al. 2014; Saxena et al. 2010; Trinh et al. 2014, 2016; Yu et al. 2010, 2014; Zheng et al. 2015, 2013]. For example, the tool S3 [Trinh et al. 2014, 2016] supports general recursively-defined predicates and uses a number of incomplete heuristics to detect unsatisfiable constraints. As another example, the tool Stranger [Yu et al. 2010, 2014] supports concatenation, replaceAll (but with both pattern and replacement strings being constants), and regular constraints, and performs widening (i.e. an overapproximation) when a concatenation operator is seen in the analysis. Despite the excellent performance of some of these solvers on several existing benchmarks, there are good reasons for designing decision procedures with stronger theoretical guarantees, e.g., in the form of decidability (perhaps accompanied by a complexity analysis). One such reason is that string constraint solving is a research area in its infancy with an insufficient range of benchmarking examples to convince us that if a string solver works well on existing benchmarks, it will also work well on future benchmarks. A theoretical result provides a kind of robustness guarantee upon which a practical solver could further improve and optimise.

Fortunately, recent years have seen the possibility of recovering some decidability of string constraint languages with multiple string operations, while retaining applicability for constraints that arise in practical symbolic execution applications. This is done by imposing syntactic restrictions

[2] A rational transduction is a transduction defined by a rational transducer, namely, a finite automaton over the alphabet $(\Sigma \cup \{\varepsilon\})^2$, where ε denotes the empty string.

including acyclicity [Abdulla et al. 2014; Barceló et al. 2013], solved form [Ganesh et al. 2012], and straight-line [Chen et al. 2018a; Holík et al. 2018; Lin and Barceló 2016]. These restrictions are known to be satisfied by many existing string constraint benchmarks, e.g., Kaluza [Saxena et al. 2010], Stranger [Yu et al. 2010], SLOG [Holík et al. 2018; Wang et al. 2016], and mutation XSS benchmarks of [Lin and Barceló 2016]. However, these results are unfortunately rather fragmented, and it is difficult to extend the comparatively limited number of supported string operations. In the following, we will elaborate this point more precisely. The acyclic logic of [Barceló et al. 2013] permits only rational transductions, in which the replaceAll function with constant pattern/replacement strings and regular constraints (but not concatenation) can be expressed. On the other hand, the acyclic logic of [Abdulla et al. 2014] permits concatenation, regular constraints, and the length function, but neither the replaceAll function nor transductions. This logic is in fact quite related to the solved-form logic proposed earlier by [Ganesh et al. 2012]. The straight-line logic of [Lin and Barceló 2016] unified the earlier logics by allowing concatenation, regular constraints, rational transductions, and length and letter-counting functions. It was pointed out by [Chen et al. 2018a] that this logic cannot express the replaceAll function with the replacement string provided as a variable, which was never studied in the context of verification and program analysis. Chen *et al.* proceeded by showing that a new straight-line logic with the more general replaceAll function and concatenation is decidable, but becomes undecidable when the length function is permitted.

Although the aforementioned results have been rather successful in capturing many string constraints that arise in applications (e.g. see the benchmarking results of [Ganesh et al. 2012] and [Holík et al. 2018; Lin and Barceló 2016]), many natural problems remain unaddressed. *To what extent can one combine these operations without sacrificing decidability?* For example, can a useful decidable logic permit the more general replaceAll, rational transductions, and concatenation at the same time? *To what extent can one introduce new string operations without sacrificing decidability?* For example, can we allow the string-reverse function (a standard library function, e.g., in Python), or more generally functions given by two-way transducers (i.e. the input head can also move to the left)? Last but not least, since there are a plethora of complex string operations, it is impossible for a solver designer to incorporate all the string operations that will be useful in all application domains. Thus, *can (and, if so, how do) we design an effective string solver that can easily be extended with user-defined string functions, while providing a strong completeness/termination guarantee?* Our goal is to provide theoretically-sound and practically implementable solutions to these problems.

Contributions. We provide two general semantic conditions (see Section 3) which together ensure decidability of path feasibility for string-manipulating programs:

(1) the conditional $R \subseteq (\Sigma^*)^k$ in each assertion admits a regular monadic decomposition, and
(2) each assignment uses a function $f : (\Sigma^*)^k \to 2^{\Sigma^*}$ whose inverse relation preserves "regularity".

Before describing these conditions in more detail, we comment on the four main features (4Es) of our decidability result: (a) *Expressive*: the two conditions are satisfied by most string constraint benchmarks (existing and new ones including those of [Holík et al. 2018; Lin and Barceló 2016; Saxena et al. 2010; Wang et al. 2016; Yu et al. 2010]) and strictly generalise several expressive and decidable constraint languages (e.g. those of [Chen et al. 2018a; Lin and Barceló 2016]), (b) *Easy*: it leads to a decision procedure that is conceptually simple (in particular, substantially simpler than many existing ones), (c) *Extensible*: it provides an extensible architecture of a string solver that allows users to easily incorporate their own user-defined functions to the solver, and (d) *Efficient*: it provides a sound basis of our new fast string solver OSTRICH that is highly competitive on string constraint benchmarks. We elaborate the details of the two aforementioned semantic conditions, and our contributions below.

The first semantic condition simply means that R can be effectively transformed into a finite union $\bigcup_{i=1}^{n}(L_i^{(1)} \times \cdots \times L_i^{(k)})$ of Cartesian products of regular languages. (Note that this is *not* the intersection/product of regular languages.) A relation that is definable in this way is often called a *recognisable relation* [Carton et al. 2006], which is one standard extension of the notion of regular languages (i.e. unary relations) to general k-ary relations. The framework of recognisable relations can express interesting conditions that might at a first glance seem beyond "regularity", e.g., $|x_1| + |x_2| \geq 3$ as can be seen below in Example 3.2. Furthermore, there are algorithms (i.e. called *monadic decompositions* in [Veanes et al. 2017]) for deciding whether a given relation represented in highly expressive symbolic representations (e.g. a *synchronised rational relation* or a *deterministic rational relation*) is recognisable and, if so, output a symbolic representation of the recognisable relation [Carton et al. 2006]. On the other hand, the second condition means that the pre-image $f^{-1}(L)$ of a regular language L under the function f is a k-ary recognisable relation. This is an expressive condition (see Section 4) satisfied by many string functions including concatenation, the string reverse function, one-way and two-way finite-state transducers, and the replaceAll function where the replacement string can contain variables. Therefore, we obtain strict generalisations of the decidable string constraint languages in [Lin and Barceló 2016] (concatenation, one-way transducers, and regular constraints) and in [Chen et al. 2018a] (concatenation, the replaceAll function, and regular constraints). In addition, many string solving benchmarks (both existing and new ones) derived from practical applications satisfy our two semantics conditions including the benchmarks of SLOG [Wang et al. 2016] with replace and replaceAll, the benchmarks of Stranger [Yu et al. 2010], ~80% of Kaluza benchmarks [Saxena et al. 2010], and the transducer benchmarks of [Holík et al. 2018; Lin and Barceló 2016]. We provide a simple and clean decision procedure (see Section 3) which propagates the regular language constraints in a *backward* manner via the regularity-preserving pre-image computation. Our semantic conditions also naturally lead to extensible architecture of a string solver: a user can easily extend our solver with one's own string functions by simply providing one's code for computing the pre-image $f^{-1}(L)$ for an input regular language L without worrying about other parts of the solver.

Having talked about the Expressive, Easy, and Extensible features of our decidability result (first three of the four Es), our decidability result does not immediately lead to an Efficient decision procedure and a fast string solver. A substantial proportion of the remaining paper is dedicated to analysing the cause of the problem and proposing ways of addressing it which are effective from both theoretical and practical standpoints.

Our hypothesis is that allowing general string relations $f : (\Sigma^*)^k \rightarrow 2^{\Sigma^*}$ (instead of just partial functions $f : \Sigma^* \rightarrow \Sigma^*$), although broadening the applicability of the resulting theory (e.g. see Figure 1), makes the constraint solving problem considerably more difficult. One reason is that propagating n regular constraints $L_1, \ldots, L_n$ backwards through a string relation $f : (\Sigma^*)^k \rightarrow 2^{\Sigma^*}$ seems to require performing a product automata construction for $\bigcap_{i=1}^{n} L_i$ before computing a recognisable relation for $f^{-1}(\bigcap_{i=1}^{n} L_i)$. To make things worse, this product construction has to be done *for practically every variable in the constraint*, each of which causes an exponential blowup. We illustrate this with a concrete example in Example 5.2. We provide a strong piece of theoretical evidence that unfortunately this is unavoidable in the worst case. More precisely, we show (see Section 4) that the complexity of the path feasibility problem with binary relations represented by one-way finite transducers (a.k.a. *binary rational relations*) and the replaceAll function (allowing a variable in the replacement string) has a NON-ELEMENTARY complexity (i.e., time/space complexity cannot be bounded by a fixed tower of exponentials) with a single level of exponentials caused by a product automata construction for each variable in the constraint. This is especially surprising since allowing either binary rational relations or the aforementioned replaceAll function results in a constraint language whose complexity is at most double exponential time and single exponential space (i.e. EXPSPACE); see [Chen et al. 2018a; Lin and Barceló 2016]. To provide further evidence of our hypothesis, we accompany this with another lower bound (also see Section 4) that the path feasibility problem has a NON-ELEMENTARY complexity for relations that are represented by two-way finite transducers (without the replaceAll function), which are possibly one of the most natural and well-studied classes of models of string relations $f : \Sigma^* \to 2^{\Sigma^*}$ (e.g. see [Alur and Deshmukh 2011; Engelfriet and Hoogeboom 2001; Filiot et al. 2013] for the model).

We propose two remedies to the problem. The first one is to allow only string functions in our constraint language. This allows one to avoid the computationally expensive product automata construction for each variable in the constraint. In fact, we show (see Section 5.1) that the NON-ELEMENTARY complexity for the case of binary rational relations and the replaceAll function can be substantially brought down to double exponential time and single exponential space (in fact, EXPSPACE-complete) if the binary rational relations are restricted to partial functions. In fact, we prove that this complexity still holds if we additionally allow the string-reverse function and the concatenation operator. The EXPSPACE complexity might still sound prohibitive, but the highly competitive performance of our new solver OSTRICH (see below) shows that this is not the case.

Our second solution (see Section 5.2) is to still allow string relations, but find an appropriate syntactic fragment of our semantic conditions that yield better computational complexity. Our proposal for such a fragment is to *restrict the use of* replaceAll *to constant replacement strings*, but allow the string-reverse function and binary rational relations. The complexity of this fragment is shown to be EXPSPACE-complete, building on the result of [Lin and Barceló 2016]. There are at least two advantages of the second solution. While string relations are supported, our algorithm reduces the problem to constraints which can be handled by the existing solver SLOTH [Holík et al. 2018] that has a reasonable performance. Secondly, the fully-fledged length constraints (e.g. $|x| = |y|$ and more generally linear arithmetic expressions on the lengths of string variables) can be incorporated into this syntactic fragment without sacrificing decidability or increasing the EXPSPACE complexity. Our experimentation and the comparison of our tool with SLOTH (see below) suggest that *our first proposed solution is to be strongly preferred when string relations are not used in the constraints.*

We have implemented our first proposed decision procedure in a new fast string solver OSTRICH[3] (*Optimistic STRIng Constraint Handler*). Our solver provides built-in support for concatenation, reverse, functional transducers (FFT), and replaceAll. Moreover, it is designed to be extensible and adding support for new string functions is a straight-forward task. We compare OSTRICH with several state-of-the-art string solving tools — including SLOTH [Holík et al. 2018], CVC4 [Liang et al. 2014], and Z3 [Berzish et al. 2017] — on a wide range of challenging benchmarks — including SLOG's replace/replaceall [Wang et al. 2016], Stranger's [Yu et al. 2010], mutation XSS [Holík et al. 2018; Lin and Barceló 2016], and the benchmarks of Kaluza that satisfy our semantic conditions (i.e. ~80% of them) [Saxena et al. 2010]. It is the only tool that was able to return an answer on all of the benchmarks we used. Moreover, it significantly outperforms SLOTH, the only tool comparable with OSTRICH in terms of theoretical guarantees and closest in terms of expressibility. It also competes well with CVC4 — a fast, but incomplete solver — on the benchmarks for which CVC4 was able to return a conclusive response. We report details of OSTRICH and empirical results in Section 6.

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 $[k] = \{1, \ldots, k\}$. For a vector $\vec{x} = (x_1, \ldots, x_n)$, let $|\vec{x}|$ denote the length of $\vec{x}$ (i.e., n) and

[3] As an aside, in contrast to an emu, an ostrich is known to be able to walk backwards, and hence the name of our solver, which propagates regular constraints in a backward direction.

$\vec{x}[i]$ denote x_i for each $i \in [n]$. Given a function $f : A \rightarrow B$ and $X \subseteq B$, we use $f^{-1}(X)$ to define the pre-image of X under f, i.e., $\{a \in A : f(a) \in X\}$.

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 $a, b, \ldots$ to denote letters from Σ and $u, v, w, \ldots$ to denote strings from Σ^*. For a string $u \in \Sigma^*$, let $|u|$ denote the *length* of u (in particular, $|\varepsilon| = 0$), moreover, for $a \in \Sigma$, let $|u|_a$ denote the number of occurrences of a in u. 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 a string $u \in \Sigma^*$, we use u^R to denote the reverse of u, that is, if $u = a_1 \cdots a_n$, then $u^R = a_n \cdots a_1$. 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 an arbitrary number of iterations of L, 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 $\Gamma = \{a_1, \ldots, a_n\} \subseteq \Sigma$, 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.

Automata models. We review some background from automata theory; for more, see [Hopcroft and Ullman 1979; Kozen 1997]. Let Σ be a finite set (called *alphabet*).

Definition 2.2 (Finite-state automata). A *(nondeterministic) finite-state automaton* (FA) over a finite alphabet Σ is a tuple $\mathcal{A} = (\Sigma, Q, q_0, F, \delta)$ where Q is a finite set of states, $q_0 \in Q$ is the initial state, $F \subseteq Q$ is a set of final states, and $\delta \subseteq Q \times \Sigma \times Q$ is the transition relation.

For an input string $w = a_1 \ldots a_n$, a *run* of $\mathcal{A}$ on w is a sequence of states $q_0, \ldots, q_n$ such that $(q_{j-1}, a_j, q_j) \in \delta$ for every $j \in [n]$. The run is said to be *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. In particular, the empty string ε is accepted by $\mathcal{A}$ iff $q_0 \in F$. The set of strings accepted by $\mathcal{A}$ is denoted by $\mathcal{L}(\mathcal{A})$, a.k.a., the language *recognised* by $\mathcal{A}$. The *size* $|\mathcal{A}|$ of $\mathcal{A}$ is defined to be $|Q|$; we will use this when we discuss computational complexity.

For convenience, we will also refer to an FA without initial and final states, that is, a pair (Q, δ), as a *transition graph.*

Operations of FAs. For an FA $\mathcal{A} = (Q, q_0, F, \delta)$, $q \in Q$ and $P \subseteq Q$, we use $\mathcal{A}(q, P)$ to denote the FA (Q, q, P, δ), that is, the FA obtained from $\mathcal{A}$ by changing the initial state and the set of final states to q and P respectively. We use $q \xrightarrow[\mathcal{A}]{w} q'$ to denote that a string w is accepted by $\mathcal{A}(q, \{q'\})$.

Given two FAs $\mathcal{A}_1 = (Q_1, q_{0,1}, F_1, \delta_1)$ and $\mathcal{A}_2 = (Q_2, q_{0,2}, F_2, \delta_2)$, the *product* of $\mathcal{A}_1$ and $\mathcal{A}_2$, denoted by $\mathcal{A}_1 \times \mathcal{A}_2$, is defined as $(Q_1 \times Q_2, (q_{0,1}, q_{0,2}), F_1 \times F_2, \delta_1 \times \delta_2)$, where $\delta_1 \times \delta_2$ is the set of tuples $((q_1, q_2), a, (q'_1, q'_2))$ such that $(q_1, a, q'_1) \in \delta_1$ and $(q_2, a, q'_2) \in \delta_2$. Evidently, we have $\mathcal{L}(\mathcal{A}_1 \times \mathcal{A}_2) = \mathcal{L}(\mathcal{A}_1) \cap \mathcal{L}(\mathcal{A}_2)$.

Moreover, let $\mathcal{A} = (Q, q_0, F, \delta)$, we define $\mathcal{A}^\pi$ as $(Q, q_f, \{q_0\}, \delta')$, where q_f is a newly introduced state not in Q and δ' comprises the transitions (q', a, q) such that $(q, a, q') \in \delta$ as well as the transitions (q_f, a, q) such that $(q, a, q') \in \delta$ for some $q' \in F$. Intuitively, $\mathcal{A}^\pi$ is obtained from $\mathcal{A} = (Q, q_0, F, \delta)$ by reversing the direction of each transition of $\mathcal{A}$ and swapping initial and final states. The new state q_f in $\mathcal{A}^\pi$ is introduced to meet the unique initial state requirement in the definition of FA. Evidently, $\mathcal{A}^\pi$ recognises the reverse language of $\mathcal{L}(\mathcal{A})$, namely, the language $\{u^R \mid u \in \mathcal{L}(\mathcal{A})\}$.

It is well-known (e.g. see [Hopcroft and Ullman 1979]) that regular expressions and FAs are expressively equivalent, and generate precisely all *regular languages*. In particular, from a regular expression, an equivalent FA can be constructed in linear time. Moreover, regular languages are closed under Boolean operations, i.e., union, intersection, and complementation.

Definition 2.3 (Finite-state transducers). Let Σ be an alphabet. A *(nondeterministic) finite transducer* (FT) T over Σ is a tuple $(\Sigma, Q, q_0, F, \delta)$, where δ is a finite subset of $Q \times \Sigma \times Q \times \Sigma^*$.

The notion of runs of FTs on an input string can be seen as a generalisation of FAs by adding outputs. More precisely, given a string $w = a_1 \ldots a_n$, a *run* of T on w is a sequence of pairs $(q_1, w'_1), \ldots, (q_n, w'_n) \in Q \times \Sigma^*$ such that for every $j \in [n]$, $(q_{j-1}, a_j, q_j, w'_j) \in \delta$. The run is said to be *accepting* if $q_n \in F$. When a run is accepting, $w'_1 \ldots w'_n$ is said to be the *output* of the run. Note that some of these w'_is could be empty strings. A word w' is said to be an output of T on w if there is an accepting run of T on w with output w'. We use $\mathcal{T}(T)$ to denote the *transduction* defined by T, that is, the relation comprising the pairs (w, w') such that w' is an output of T on w.

We remark that an FT usually defines a *relation*. We shall speak of *functional transducers*, i.e., transducers that define functions instead of relations. (For instance, deterministic transducers are always functional.) We will use FFT to denote the class of functional transducers.

To take into consideration the outputs of transitions, we define the *size* $|T|$ of T as the sum of the sizes of transitions in T, where the size of a transition (q, a, q', w') is defined as $|w'| + 1$.

Example 2.4. We give an example FT for the function **escapeString**, which backslash-escapes every occurrence of ' and ". The FT has a single state, i.e., $Q = \{q_0\}$ and the transition relation δ comprises (q_0, ℓ, q_0, ℓ) for each $\ell \neq$ ' or ", $(q_0,$ ' $, q_0,$ \' $)$, $(q_0,$ " $, q_0,$ \" $)$, and the final state $F = \{q_0\}$. We remark that this FT is functional. □

Computational Complexity. In this paper, we will use computational complexity theory to provide evidence that certain (automata) operations in our generic decision procedure are unavoidable. 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), EXPSPACE (problems solvable in exponential space and thus in double exponential time), and NON-ELEMENTARY (problems not a member of the class ELEMENTARY, where ELEMENTARY comprises elementary recursive functions, which is the union of the complexity classes EXPTIME, 2-EXPTIME, 3-EXPTIME, . . ., or alternatively, the union of the complexity classes EXPSPACE, 2-EXPSPACE, 3-EXPSPACE, . . .). 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 SEMANTIC CONDITIONS AND A GENERIC DECISION PROCEDURE

Recall that we consider symbolic executions of string-manipulating programs defined by the rules

$$S ::= \qquad y := f(x_1, \ldots, x_r) \mid \mathbf{assert}(g(x_1, \ldots, x_r)) \mid S; S \tag{3}$$

where $f : (\Sigma^*)^r \to 2^{\Sigma^*}$ is a *nondeterministic* partial string function and $g \subseteq (\Sigma^*)^r$ is a string relation. Without loss of generality, we assume that symbolic executions are in Static Single Assignment (SSA) form.[4]

In this section, we shall provide two general semantic conditions for symbolic executions. The main result is that, whenever the symbolic execution generated by (3) satisfies these two conditions, the path feasibility problem is decidable. We first define the concept of recognisable relations which, intuitively, are simply a finite union of Cartesian products of regular languages.

Definition 3.1 (Recognisable relations). An r-ary relation $R \subseteq \Sigma^* \times \cdots \times \Sigma^*$ is *recognisable* if $R = \bigcup_{i=1}^{n} L_1^{(i)} \times \cdots \times L_r^{(i)}$ where $L_j^{(i)}$ is regular for each $j \in [r]$. A *representation* of a recognisable relation $R = \bigcup_{i=1}^{n} L_1^{(i)} \times \cdots \times L_r^{(i)}$ is $(\mathcal{A}_1^{(i)}, \ldots, \mathcal{A}_r^{(i)})_{1 \le i \le n}$ such that each $\mathcal{A}_j^{(i)}$ is an FA with $\mathcal{L}(\mathcal{A}_j^{(i)}) = L_j^{(i)}$. The tuples $(\mathcal{A}_1^{(i)}, \ldots, \mathcal{A}_r^{(i)})$ are called the *disjuncts* of the representation and the FAs $\mathcal{A}_j^{(i)}$ are called the *atoms* of the representation.

We remark that the recognisable relation is more expressive than it appears to be. For instance, it can be used to encode some special length constraints, as demonstrated in Example 3.2.

Example 3.2. Let us consider the relation $|x_1| + |x_2| \geq 3$ where x_1 and x_2 are strings over the alphabet Σ. Although syntactically $|x_1| + |x_2| \geq 3$ is a length constraint, it indeed defines a recognisable relation. To see this, $|x_1| + |x_2| \geq 3$ is equivalent to the disjunction of $|x_1| \geq 3$, $|x_1| \geq 2 \wedge |x_2| \geq 1$, $|x_1| \geq 1 \wedge |x_2| \geq 2$, and $|x_2| \geq 3$, where each disjunct describes a cartesian product of regular languages. For instance, in $|x_1| \geq 2 \wedge |x_2| \geq 1$, $|x_1| \geq 2$ requires that x_1 belongs to the regular language $\Sigma \cdot \Sigma^+$, while $|x_2| \geq 1$ requires that x_2 belongs to the regular language Σ^+. □

The equality binary predicate $x_1 = x_2$ is a standard non-example of recognisable relations; in fact, expressing $x_1 = x_2$ as a union $\bigcup_{i \in I} L_i \times H_i$ of products requires us to have $|L_i| = |H_i| = 1$, which in turn forces us to have an infinite index set I.

The first semantic condition, *Regular Monadic Decomposition* is stated as follows.

RegMonDec: For each assertion $\mathbf{assert}(g(x_1, \ldots, x_r))$ in S, g is a recognisable relation, a representation of which, in terms of Definition 3.1, is effectively computable.

When $r = 1$, the **RegMonDec** condition requires that $g(x_1)$ is regular and may be given by an FA $\mathcal{A}$, in which case $x_1 \in \mathcal{L}(\mathcal{A})$.

The second semantic condition concerns the pre-images of string operations. A string operation $f(x_1, \ldots, x_r)$ with r parameters ($r \geq 1$) gives rise to a relation $R_f \subseteq (\Sigma^*)^r \times \Sigma^*$. Let $L \subseteq \Sigma^*$. The *pre-image* of L under f, denoted by $\mathsf{Pre}_{R_f}(L)$, is

$$\{(w_1, \ldots, w_r) \in (\Sigma^*)^r \mid \exists w.\ w \in f(w_1, \ldots, w_r) \text{ and } w \in L\}.$$

For brevity, we use $\mathsf{Pre}_{R_f}(\mathcal{A})$ to denote $\mathsf{Pre}_{R_f}(\mathcal{L}(\mathcal{A}))$ for an FA $\mathcal{A}$. The second semantic condition, i.e. the inverse relation of f preserves regularity, is formally stated as follows.

RegInvRel: For each operation f in S and each FA $\mathcal{A}$, $\mathsf{Pre}_{R_f}(\mathcal{A})$ is a recognisable relation, a representation of which (Definition 3.1), can be effectively computed from $\mathcal{A}$ and f.

[4] Each symbolic execution can be turned into the SSA form by using a new variable on the left-hand-side of each assignment.

When $r = 1$, this **RegInvRel** condition would state that the pre-image of a regular language under the operation f is *effectively regular*, i.e. an FA can be computed to represent the pre-image of the regular language under f.

Example 3.3. Let $\Sigma = \{a, b\}$. Consider the string function $f(x_1, x_2) = a^{|x_1|_a + |x_2|_a} b^{|x_1|_b + |x_2|_b}$. (Recall that $|x|_a$ denotes the number of occurrences of a in x.) We can show that for each FA $\mathcal{A}$, $\mathsf{Pre}_{R_f}(\mathcal{A})$ is a recognisable relation. Let $\mathcal{A}$ be an FA. W.l.o.g. we assume that $\mathcal{L}(\mathcal{A}) \subseteq a^* b^*$. It is easy to observe that $\mathcal{L}(\mathcal{A})$ is a finite union of the languages $\{a^{c_1 p + c_2} b^{c_1' p' + c_2'} \mid p \in \mathbb{N}, p' \in \mathbb{N}\}$, where c_1, c_2, c_1', c_2' are natural number constants. Therefore, to show that $\mathsf{Pre}_{R_f}(\mathcal{A})$ is a recognisable relation, it is sufficient to show that $\mathsf{Pre}_{R_f}(\{a^{c_1 p + c_2} b^{c_1' p' + c_2'} \mid p \in \mathbb{N}, p' \in \mathbb{N}\})$ is a recognisable relation.

Let us consider the typical situation that $c_1 \neq 0$ and $c_1' \neq 0$. Then $\mathsf{Pre}_{R_f}(\{a^{c_1 p + c_2} b^{c_1' p' + c_2'} \mid p \in \mathbb{N}, p' \in \mathbb{N}\})$ is the disjunction of $L_1^{(i,i')} \times L_2^{(j,j')}$ for $i, j, i', j' \in \mathbb{N}$ with $i + j = c_2$, and $i' + j' = c_2'$, where $L_1^{(i,i')} = \{u \in \Sigma^* \mid |u|_a \geq i, |u|_a \equiv i \bmod c_1, |u|_b \geq i', |u|_b \equiv i' \bmod c_1'\}$, $L_2^{(j,j')} = \{v \in \Sigma^* \mid |v|_a \geq j, |v|_a \equiv j \bmod c_1, |v|_b \geq j', |v|_b \equiv j' \bmod c_1'\}$. Evidently, $L_1^{(i,i')}$ and $L_2^{(j,j')}$ are regular languages. Therefore, $\mathsf{Pre}_{R_f}(\{a^{c_1 p + c_2} b^{c_1' p' + c_2'} \mid p \in \mathbb{N}, p' \in \mathbb{N}\})$ is a finite union of cartesian products of regular languages, and thus a recognisable relation. □

Not every string operation satisfies the **RegInvRel** condition, as demonstrated by Example 3.4.

Example 3.4. Let us consider the string function f on the alphabet $\{0, 1\}$ that transforms the unary representations of natural numbers into their binary representations, namely, $f(1^n) = b_0 b_1 \dots b_m$ such that $n = 2^m b_0 + \cdots + 2 b_{m-1} + b_m$ and $b_0 = 1$. For instance, $f(1^4) = 100$. We claim that f does not satisfy the **RegInvRel** condition. To see this, consider the regular language $L = \{10^i \mid i \in \mathbb{N}\}$. Then $\mathsf{Pre}_{R_f}(L)$ comprises the strings 1^{2^j} with $j \in \mathbb{N}$, which is evidently non-regular. Incidentally, this is an instance of the well-known Cobham's theorem (cf. [Pippenger 2010]) that the sets of numbers definable by finite automata in unary are strictly subsumed by the sets of numbers definable by finite automata in binary. □

We are ready to state the main result of this section.

Theorem 3.5. *The path feasibility problem is decidable for symbolic executions satisfying the* ***RegMonDec*** *and* ***RegInvRel*** *conditions.*

Proof of Theorem 3.5. We present a nondeterministic decision procedure from which the theorem follows.

Let S be a symbolic execution, $y := f(\vec{x})$ (where $\vec{x} = x_1, \dots, x_r$) be the last assignment in S, and $\rho := \{g_1(\vec{z}_1), \dots, g_s(\vec{z}_s)\}$ be the set of all constraints in assertions of S that involve y (i.e. y occurs in $\vec{z}_i$ for all $i \in [s]$). For each $i \in [s]$, let $\vec{z}_i = (z_{i,1}, \dots, z_{i,\ell_i})$. Then by the **RegMonDec** assumption, g_i is a recognisable relation and a representation of it, say $\left(\mathcal{A}_{i,1}^{(j)}, \dots, \mathcal{A}_{i,\ell_i}^{(j)}\right)_{1 \leq j \leq n_i}$ with $n_i \geq 1$, can be effectively computed.

For each $i \in [s]$, we nondeterministically choose one tuple $(\mathcal{A}_{i,1}^{(j_i)}, \dots, \mathcal{A}_{i,\ell_i}^{(j_i)})$ (where $1 \leq j_i \leq n_i$), and for all $i \in [s]$, replace $\mathbf{assert}(g_i(\vec{z}_i))$ in S with $\mathbf{assert}(z_{i,1} \in \mathcal{A}_{i,1}^{(j_i)}); \dots; \mathbf{assert}(z_{i,\ell_i} \in \mathcal{A}_{i,\ell_i}^{(j_i)})$. Let S' denote the resulting program.

We use σ to denote the set of all the FAs $\mathcal{A}_{i,i'}^{(j_i)}$ such that $1 \leq i \leq s$, $1 \leq i' \leq \ell_i$, and $\mathbf{assert}(y \in \mathcal{A}_{i,i'}^{(j_i)})$ occurs in S'. We then compute the product FA $\mathcal{A}$ from FAs $\mathcal{A}_{i,i'}^{(j_i)} \in \sigma$ such that $\mathcal{L}(\mathcal{A})$ is the intersection of the languages defined by FAs in σ. By the **RegInvRel** assumption, $g' = \mathsf{Pre}_{R_f}(\mathcal{A})$ is a recognisable relation and a representation of it can be effectively computed.

Let S'' be the symbolic execution obtained from S' by (1) removing $y := f(\vec{x})$ along with all assertions involving y (i.e. the assertions $\mathbf{assert}(y \in \mathcal{A}_{i,i'}^{(j_i)})$ for $\mathcal{A}_{i,i'}^{(j_i)} \in \sigma$), (2) and adding the assertion $\mathbf{assert}(g'(x_1, \ldots, x_r))$.

It is straightforward to verify that S is path-feasible iff there is a nondeterministic choice resulting in S' that is path-feasible, moreover, S' is path feasible iff S'' is path-feasible. Evidently, S'' has one less assignment than S. Repeating these steps, the procedure will terminate when S becomes a conjunction of assertions on input variables, the feasibility of which can be checked via language nonemptiness checking of FAs. To sum up, the correctness of the (nondeterministic) procedure follows since the path-feasibility is preserved for each step, and the termination is guaranteed by the finite number of assignments. □

Let us use the following example to illustrate the generic decision procedure.

Example 3.6. Consider the symbolic execution

$$\mathbf{assert}(x \in \mathcal{A}_0);\ y_1 := f(x);\ z := y_1 \circ y_2;\ \mathbf{assert}(y_1 \in \mathcal{A}_1);\ \mathbf{assert}(y_2 \in \mathcal{A}_2);\ \mathbf{assert}(z \in \mathcal{A}_3)$$

where $\mathcal{A}_0, \mathcal{A}_1, \mathcal{A}_2, \mathcal{A}_3$ are FAs illustrated in Figure 2, and $f : \Sigma^* \to 2^{\Sigma^*}$ is the function mentioned in Section 1 that nondeterministically outputs a substring delimited by -. At first, we remove the assignment $z = y_1 \circ y_2$ as well as the assertion $\mathbf{assert}(z \in \mathcal{A}_3)$. Moreover, since the pre-image of $\circ$ under $\mathcal{A}_3$, denoted by g, is a recognisable relation represented by $(\mathcal{A}_3(q_0, \{q_i\}), \mathcal{A}_3(q_i, \{q_0\}))_{0 \le i \le 2}$, we add the assertion $\mathbf{assert}(g(y_1, y_2))$, and get following program

$$\mathbf{assert}(x \in \mathcal{A}_0);\ y_1 := f(x);\ \mathbf{assert}(y_1 \in \mathcal{A}_1);\ \mathbf{assert}(y_2 \in \mathcal{A}_2);\ \mathbf{assert}(g(y_1, y_2)).$$

To continue, we nondeterministically choose one tuple, say $(\mathcal{A}_3(q_0, \{q_1\}), \mathcal{A}_3(q_1, \{q_0\}))$, from the representation of g, and replace $\mathbf{assert}(g(y_1, y_2))$ with $\mathbf{assert}(y_1 \in \mathcal{A}_3(q_0, \{q_1\}))$; $\mathbf{assert}(y_2 \in \mathcal{A}_3(q_1, \{q_0\}))$, and get the program

$$\mathbf{assert}(x \in \mathcal{A}_0);\ y_1 := f(x);\ \mathbf{assert}(y_1 \in \mathcal{A}_1);\ \mathbf{assert}(y_2 \in \mathcal{A}_2);$$
$$\mathbf{assert}(y_1 \in \mathcal{A}_3(q_0, \{q_1\}));\ \mathbf{assert}(y_2 \in \mathcal{A}_3(q_1, \{q_0\})).$$

Let σ be $\{\mathcal{A}_1, \mathcal{A}_3(q_0, \{q_1\})\}$, the set of FAs occurring in the assertions for y_1 in the above program. Compute the product $\mathcal{A}' = \mathcal{A}_1 \times \mathcal{A}_3(q_0, \{q_1\})$ and $\mathcal{A}'' = \mathrm{Pre}_{R_f}(\mathcal{A}')$ (see Figure 2).

Then we remove $y_1 := f(x)$, as well as the assertions that involve y_1, namely, $\mathbf{assert}(y_1 \in \mathcal{A}_1)$ and $\mathbf{assert}(y_1 \in \mathcal{A}_3(q_0, \{q_1\}))$, and add the assertion $\mathbf{assert}(x \in \mathcal{A}'')$, resulting in the program

$$\mathbf{assert}(x \in \mathcal{A}_0);\ \mathbf{assert}(y_2 \in \mathcal{A}_2);\ \mathbf{assert}(y_2 \in \mathcal{A}_3(q_1, \{q_0\}));\ \mathbf{assert}(x \in \mathcal{A}'').$$

It is not hard to see that $\text{-}\,a\,\text{-} \in \mathcal{L}(\mathcal{A}_0) \cap \mathcal{L}(\mathcal{A}'')$ and $abb \in \mathcal{L}(\mathcal{A}_2) \cap \mathcal{L}(\mathcal{A}_3(q_1, \{q_0\}))$. Then the assignment $x = \text{-}\,a\,\text{-}$, $y_1 = a$, $y_2 = abb$, and $z = aabb$ witnesses the path feasibility of the original symbolic execution. □

Remark 3.7. *Theorem 3.5 gives two semantic conditions which are sufficient to render the path feasibility problem decidable. A natural question, however, is how to check whether a given symbolic execution satisfies the two semantic conditions. The answer to this meta-question highly depends on the classes of string operations and relations under consideration. Various classes of relations which admit finite representations have been studied in the literature. They include, in an ascending order of expressiveness, recognisable relations, synchronous relations, deterministic rational relations, and rational relations, giving rise to a strict hierarchy. (We note that slightly different terminologies tend to be used in the literature, for instance, synchronous relations in [Carton et al. 2006] are called regular relations in [Barceló et al. 2013] which are also known as automatic relations, synchronised rational relations, etc. One may consult the survey [Choffrut 2006] and [Carton et al. 2006].) It is known [Carton et al. 2006] that determining whether a given deterministic rational relation is recognisable is decidable*

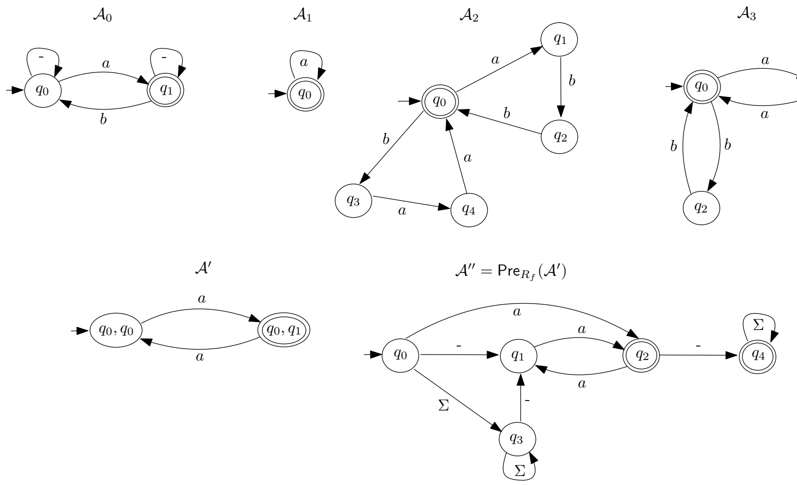

Fig. 2. $\mathcal{A}_0, \mathcal{A}_1, \mathcal{A}_2, \mathcal{A}_3, \mathcal{A}', \mathsf{Pre}_{R_f}(\mathcal{A}')$, where $\Sigma = \{a, b, \text{-}\}$

(for binary relations, this can be done in doubly exponential time), and deciding whether a synchronous relation is recognisable can be done in exponential time [Carton et al. 2006]. Similar results are also mentioned in [Benedikt et al. 2003; Libkin 2003].

By these results, one can check, for a given symbolic execution where the string relations in the assertion and the relations induced by the string operation are all deterministic rational *relations, whether it satisfies the two semantic conditions. Hence, one can check algorithmically whether Theorem 3.5 is applicable.*

□

4 AN EXPRESSIVE LANGUAGE SATISFYING THE SEMANTIC CONDITIONS

Section 3 has identified general *semantic* conditions under which the *decidability* of the path feasibility problem can be attained. Two questions naturally arise:

(1) How general are these semantic conditions? In particular, do string functions commonly used in practice satisfy these semantic conditions?
(2) What is the computational complexity of checking path feasibility?

The next two sections will be devoted to answering these questions.

For the first question, we shall introduce a *syntactically* defined string constraint language SL, which includes general string operations such as the replaceAll function and those definable by two-way transducers, as well as recognisable relations. [Here, SL stands for "straight-line" because our work generalises the straight-line logics of [Chen et al. 2018a; Lin and Barceló 2016].] We first recap the replaceAll function that allows a general (i.e. variable) replacement string [Chen et al. 2018a]. Then we give the definition of two-way transducers whose special case (i.e. one-way transducers) has been given in Section 2.

4.1 The replaceAll function and two-way transducers

The replaceAll function has three parameters: the first parameter is the *subject* string, the second parameter is a *pattern* that is a regular expression, and the third parameter is the *replacement* string. For the semantics of replaceAll function, in particular when the pattern is a regular expression, we adopt the *leftmost and longest* matching. For instance, $\mathsf{replaceAll}(aababaab, (ab)^+, c) = ac \cdot$

$\mathsf{replaceAll}(aab, (ab)^+, c) = acac$, since the leftmost and longest matching of $(ab)^+$ in $aababaab$ is $abab$. Here we require that the language defined by the pattern parameter does *not* contain the empty string, in order to avoid the troublesome definition of the semantics of the matching of the empty string. We refer the reader to [Chen et al. 2018a] for the formal semantics of the replaceAll function. To be consistent with the notation in this paper, for each regular expression e, we define the string function $\mathsf{replaceAll}_e : \Sigma^* \times \Sigma^* \to \Sigma^*$ such that for $u, v \in \Sigma^*$, $\mathsf{replaceAll}_e(u, v) = \mathsf{replaceAll}(u, e, v)$, and we write $\mathsf{replaceAll}(x, e, y)$ as $\mathsf{replaceAll}_e(x, y)$.

As in the one-way case, we start with a definition of two-way finite-state automata.

Definition 4.1 (Two-way finite-state automata). A *(nondeterministic) two-way finite-state automaton* (2FA) over a finite alphabet Σ is a tuple $\mathcal{A} = (\Sigma, \triangleright, \triangleleft, Q, q_0, F, \delta)$ where Q, q_0, F are as in FAs, $\triangleright$ (resp. $\triangleleft$) is a left (resp. right) input tape end marker, and the transition relation $\delta \subseteq Q \times \overline{\Sigma} \times \{-1, 1\} \times Q$, where $\overline{\Sigma} = \Sigma \cup \{\triangleright, \triangleleft\}$. Here, we assume that there are no transitions that take the head of the tape past the left/right end marker (i.e. $(p, \triangleright, -1, q), (p, \triangleleft, 1, q) \notin \delta$ for every $p, q \in Q$).

Whenever they can be easily understood, we will not mention Σ, $\triangleright$, and $\triangleleft$ in $\mathcal{A}$.

The notion of runs of 2FA on an input string is exactly the same as that of Turing machines on a read-only input tape. More precisely, for a string $w = a_1 \dots a_n$, a *run* of $\mathcal{A}$ on w is a sequence of pairs $(q_0, i_0), \dots, (q_m, i_m) \in Q \times [0, n+1]$ defined as follows. Let $a_0 = \triangleright$ and $a_{n+1} = \triangleleft$. The following conditions then have to be satisfied: $i_0 = 0$, and for every $j \in [0, m-1]$, we have $(q_j, a_{i_j}, dir, q_{j+1}) \in \delta$ and $i_{j+1} = i_j + dir$ for some $dir \in \{-1, 1\}$.

The run is said to be *accepting* if $i_m = n + 1$ and $q_m \in F$. A string w is *accepted* by $\mathcal{A}$ if there is an accepting run of $\mathcal{A}$ on w. The set of strings accepted by $\mathcal{A}$ is denoted by $\mathcal{L}(\mathcal{A})$, a.k.a., the language *recognised* by $\mathcal{A}$. The *size* $|\mathcal{A}|$ of $\mathcal{A}$ is defined to be $|Q|$; this will be needed when we talk about computational complexity.

Note that an FA can be seen as a 2FA such that $\delta \subseteq Q \times \overline{\Sigma} \times \{1\} \times Q$, with the two end markers $\triangleright, \triangleleft$ omitted. 2FA and FA recognise precisely the same class of languages, i.e., *regular languages.* The following result is standard and can be found in textbooks on automata theory (e.g. [Hopcroft and Ullman 1979]).

Proposition 4.2. *Every 2FA $\mathcal{A}$ can be transformed in exponential time into an equivalent FA of size $2^{O(|\mathcal{A}| \log |\mathcal{A}|)}$.*

Definition 4.3 (Two-way finite-state transducers). Let Σ be an alphabet. A *nondeterministic two-way finite transducer* (2FT) T over Σ is a tuple $(\Sigma, \triangleright, \triangleleft, Q, q_0, F, \delta)$, where Σ, Q, q_0, F are as in FTs, and $\delta \subseteq Q \times \overline{\Sigma} \times \{-1, 1\} \times Q \times \Sigma^*$, satisfying the syntactical constraints of 2FAs, and the additional constraint that the output must be ϵ when reading $\triangleright$ or $\triangleleft$. Formally, for each transition $(q, \triangleright, dir, q', w)$ or $(q, \triangleleft, dir, q', w)$ in δ, we have $w = \epsilon$.

The notion of runs of 2FTs on an input string can be seen as a generalisation of 2FAs by adding outputs. More precisely, given a string $w = a_1 \dots a_n$, a *run* of T on w is a sequence of tuples $(q_0, i_0, w'_0), \dots, (q_m, i_m, w'_m) \in Q \times [0, n+1] \times \Sigma^*$ such that, if $a_0 = \triangleright$ and $a_{n+1} = \triangleleft$, we have $i_0 = 0$, and for every $j \in [0, m-1]$, $(q_j, a_{i_j}, dir, q_{j+1}, w'_j) \in \delta$, $i_{j+1} = i_j + dir$ for some $dir \in \{-1, 1\}$, and $w'_0 = w'_m = \varepsilon$. The run is said to be *accepting* if $i_m = n + 1$ and $q_m \in F$. When a run is accepting, $w'_0 \dots w'_m$ is said to be the *output* of the run. Note that some of these w'_is could be empty strings. A word w' is said to be an output of T on w if there is an accepting run of T on w with output w'. We use $\mathcal{T}(T)$ to denote the *transduction* defined by T, that is, the relation comprising the pairs (w, w') such that w' is an output of T on w.

Note that an FT over Σ is a 2FT such that $\delta \subseteq Q \times \overline{\Sigma} \times \{1\} \times Q \times \Sigma^*$, with the two endmarkers $\triangleright, \triangleleft$ omitted.

Example 4.4. We give an example of 2FT for the function $f(w) = ww^R$. The transducer has three states $Q = \{q_0, q_1, q_2\}$, and the transition relation δ comprises $(q_0, \ell, 1, q_0, \ell)$ for $\ell \in \Sigma$, $(q_0, \rhd, 1, q_0, \epsilon)$, $(q_0, \lhd, -1, q_1, \epsilon)$, $(q_1, \ell, -1, q_1, \ell)$ for $\ell \in \Sigma$, $(q_1, \rhd, 1, q_2, \epsilon)$, $(q_2, \ell, 1, q_2, \epsilon)$ for $\ell \in \Sigma$. The final state $F = \{q_2\}$. □

4.2 The constraint language SL

The constraint language SL is defined by the following rules,

$$S ::= \quad z := x \circ y \mid z := \mathsf{replaceAll}_e(x, y) \mid y := \mathsf{reverse}(x) \mid y := T(x) \mid \mathbf{assert}(R(\vec{x})) \mid S; S \qquad (4)$$

where $\circ$ is the string concatenation operation which concatenates two strings, e is a regular expression, reverse is the string function which reverses a string, T is a 2FT, and R is a recognisable relation represented by a collection of tuples of FAs.

For the convenience of Section 5, for a class of string operations $\mathscr{O}$, we will use SL[$\mathscr{O}$] to denote the fragments of SL that only use the string operations from $\mathscr{O}$. Moreover, we will use sreplaceAll to denote the special case of the $\mathsf{replaceAll}_e$ function where the replacement parameters are restricted to be *string constants*. Note that, according to the result in [Chen et al. 2018a], an instance of the sreplaceAll function $\mathsf{replaceAll}_e(x, u)$ with e a regular expression and u a string constant can be captured by FTs. However, such a transformation incurs an exponential blow-up. We also remark that we do not present SL in the most succinct form. For instance, it is known that the concatenation operation can be simulated by the replaceAll function, specifically, $z = x \circ y \equiv z' = \mathsf{replaceAll}_a(ab, x) \wedge z = \mathsf{replaceAll}_b(z', y)$, where a, b are two fresh letters. Moreover, it is evident that the reverse function is subsumed by 2FTs.

We remark that SL is able to encode some string functions with multiple (greater than two) arguments by transducers and repeated use of replaceAll, which is practically convenient particularly for user-defined functions.

The following theorem answers the two questions raised in the beginning of this section.

Theorem 4.5. *The path feasibility problem of* SL *is decidable with a non-elementary lower-bound.*

To invoke the result of the previous section, we have the following proposition.

Proposition 4.6. *The* SL *language satisfies the two semantic conditions* ***RegMonDec*** *and* ***RegInvRel****.*

Proof. It is sufficient to show that the $\mathsf{replaceAll}_e$ functions and the string operations defined by 2FTs satisfy the **RegInvRel** condition.

The fact that $\mathsf{replaceAll}_e$ for a given regular expression e satisfies the **RegInvRel** condition was shown in [Chen et al. 2018a].

That the pre-image of an 2FT T under a regular language defined by an FA $\mathcal{A}$ is effectively regular is folklore. Let $T = (Q, q_0, F, \delta)$ be a 2FT and $\mathcal{A} = (Q', q'_0, F', \delta')$ be an FA. Then $\mathsf{Pre}_{\mathcal{T}(T)}(\mathcal{A})$ is the regular language defined by the 2FA $\mathcal{A}' = (Q \times Q', (q_0, q'_0), F \times F', \delta'')$, where δ'' comprises the tuples $((q_1, q'_1), a, (q_2, q'_2))$ such that there exists $w \in \Sigma^*$ satisfying that $(q_1, a, q_2, w) \in \delta$ and $q'_1 \xrightarrow[\mathcal{A}]{w} q'_2$. From Proposition 4.2, an equivalent FA can be built from $\mathcal{A}'$ in exponential time. □

From Proposition 4.6 and Theorem 3.5, the path feasibility problem of SL is *decidable*.

To address the complexity (viz. the second question raised at the beginning of this section), we show that the path feasibility problem of SL is non-elementary.

Proposition 4.7. *The path feasibility problem of the following two fragments is* non-elementary*:* SL *with 2FTs, and* SL *with FTs+*replaceAll.

For each n we reduce from a tiling problem that is hard for n-EXPSPACE. For this we need to use large numbers that act as indices. Similar encodings of large numbers appear in the study of higher-order programs (e.g. [Cachat and Walukiewicz 2007; Jones 2001]) except quite different machinery is needed to enforce the encoding. The complete reduction is given in the full version of this article [Chen et al. 2018b], with some intuition given here.

A *tiling problem* consists of a finite set of tiles Θ as well as horizontal and vertical tiling relations $H, V \subseteq \Theta \times \Theta$. Given a tiling *corridor* of a certain width, as well as initial and final tiles $t_I, t_F \in \Theta$ the task is to find a tiling where the first (resp. last) tile of the first (resp. last) row is t_I (resp. t_F), and horizontally (resp. vertically) adjacent tiles t, t' have $(t, t') \in H$ (resp. V). Corridor width can be considered equivalent to the space of a Turing machine. We will consider problems where the corridor width is $2^{\cdot^{\cdot^{2^m}}}$ where the height of the stack of exponentials is n. E.g. when n is 0 the width is m, when n is 1 the width is 2^m, when n is 2 the width is 2^{2^m} and so on. Solving tiling problems of width $2^{\cdot^{\cdot^{2^m}}}$ is complete for the same amount of space.

Solving a tiling problem of corridor width m can be reduced to checking whether a 2FT of size polynomial in m and the number of tiles can output a specified symbol $\top$. Equivalently, we could use a 2FA. A solution is a word

$$t_{1,1}t_{1,2} \dots t_{1,m}\# \dots \# t_{h,1}t_{h,2} \dots t_{h,m}$$

where # separates rows. The 2FT performs $m + 1$ passes. During the first pass it checks that the tiling begins with t_I, ends with t_F, and $\left(t_{i,j}, t_{i,j+1}\right) \in H$ for all $1 \leq j < m$. In m more passes we verify that V is obeyed; the jth pass verifies the jth column.

Now consider two 2FTs and a tiling problem of width 2^m. Intuitively, we precede each tile with its column number in m binary bits. That is

$$0 \dots 00\ t_{1,1}\ 0 \dots 01\ t_{1,2}\ \dots 1 \dots 11\ t_{1,2^m}\ \# \dots \# 0 \dots 00\ t_{m,1}\ 0 \dots 01\ t_{m,2}\ \dots 1 \dots 11\ t_{m,2^m}\ .$$

The first 2FT checks the solution similarly to the width m problem, but needs to handle the large width when checking V. For this it will use a second 2FT. For each column, the first 2FT nondeterministically selects all the tiles in this column (verifying V on-the-fly). The addresses of the selected tiles are output to the second 2FT which checks that they are equal. The first 2FT goes through a non-deterministic number of such passes and the second 2FT enforces that there are 2^m of them (in column order). To do this, the second 2FT checks that after the addresses of the i-th column are output by the first 2FT, then the addresses of the $(i + 1)$-th column are output next. Length m binary numbers are checked similarly to width m tiling problems.

With another 2FT we can increase the corridor width by another exponential. For doubly-exponential numbers, we precede each tile with a binary sequence of exponential length. For this we precede each bit with another binary sequence, this time of length m. The first 2FT outputs queries to the second, which outputs queries to the third 2FT, each time removing one exponential. With $(n + 1)$ 2FT, we can encode tiling problems over an n-fold exponentially wide corridor.

The same proof strategy can be used for FTs+replaceAll. The 2FTs used in the proof above proceed by running completely over the word and producing some output, then silently moving back to the beginning of the word. An arbitrary number of passes are made in this way. We can simulate this behaviour using FTs and replaceAll.

To simulate $y := T(x)$ for a 2FT T making an arbitrary number of passes over the contents of a variable x, as above, we use fresh variables x_1 and x_2, and an automaton $\mathcal{A}_a$ recognising $(a\natural)^*$ for some arbitrary character a and delimiter $\natural$ not used elsewhere. With these we use the constraint

$$\textbf{assert}(x_1 \in \mathcal{A}_a); x_2 := \mathsf{replaceAll}_a(x_1, x); y := T'(x_2)$$

where T' simulates T in the forwards direction, and simulates (simply by changing state) a silent return to the beginning of the word when reading $\natural$. It can be seen that x_2 contains an arbitrary number of copies of x, separated by $\natural$, hence T' simulates T.

It was stated in Section 1 that the NON-ELEMENTARY complexity will be caused by repeated product constructions. These product constructions are not obvious here, but are hidden in the treatment of $\mathsf{replaceAll}_a$. This treatment is elaborated on in Section 6.2.2. The key point is that to show $\mathsf{replaceAll}_a$ satisfies **RegInvRel** one needs to produce a constraint on x that is actually the product of several automata.

5 MORE "TRACTABLE" FRAGMENTS

In this section, we show that the NON-ELEMENTARY lower-bound of the preceding section should not be read too pessimistically. As we demonstrate in this section, the complexity of the path feasibility problem can be dramatically reduced (EXPSPACE-complete) for the following two fragments,

- SL[∘, replaceAll, reverse, FFT], where 2FTs in SL are restricted to be *one-way and functional*,
- SL[∘, sreplaceAll, reverse, FT], where the *replacement* parameter of the replaceAll function is restricted to be a string *constant*, and 2FTs are restricted to be *one-way* (but *not* necessarily functional).

These two fragments represent the most practical usage of string functions. In particular, instead of very general two-way transducers, one-way transducers are commonly used to model, for instance, browser transductions [Lin and Barceló 2016].

5.1 The fragment SL[∘, replaceAll, reverse, FFT]

The main result of this subsection is stated in the following theorem.

THEOREM 5.1. *The path feasibility of* SL[∘, replaceAll, reverse, *FFT] is* EXPSPACE*-complete.*

To show the upper bound of Theorem 5.1, we will refine the (generic) decision procedure in Section 3 in conjunction with a careful complexity analysis. The crucial idea is to *avoid the product construction* before each pre-image computation in the algorithm given in the proof of Theorem 3.5. This is possible now since all string operations in SL[∘, replaceAll, reverse, FFT] are (partial) functions, and regular constraints are distributive with respect to the pre-image of string (partial) functions.

FACT 1. *For every string (partial) function $f : (\Sigma^*)^r \rightarrow \Sigma^*$ and regular languages L_1 and L_2, it holds that $Pre_{R_f}(L_1 \cap L_2) = Pre_{R_f}(L_1) \cap Pre_{R_f}(L_2)$.*

To see this, suppose $\vec{u} \in \mathsf{Pre}_{R_f}(L_1) \cap \mathsf{Pre}_{R_f}(L_2)$. Then $\vec{u} \in \mathsf{Pre}_{R_f}(L_1)$ and $\vec{u} \in \mathsf{Pre}_{R_f}(L_2)$. Therefore, there are $v_1 \in L_1, v_2 \in L_2$ such that $(\vec{u}, v_1) \in R_f$ and $(\vec{u}, v_2) \in R_f$. Since f is a (partial) function, it follows that $v_1 = v_2 \in L_1 \cap L_2$, thus $\vec{u} \in \mathsf{Pre}_{R_f}(L_1 \cap L_2)$. This equality does *not* hold in general if f is *not* functional, as shown by the following example.

Example 5.2. Let $\text{-} \in \Sigma$ and $f : \Sigma^* \rightarrow 2^{\Sigma^*}$ be the nondeterministic function mentioned in the introduction (see Figure 1) that nondeterministically outputs a substring delimited by −. Moreover, let a, b be two distinct letters from $\Sigma \setminus \{\text{-}\}$, $L_1 = a(\Sigma \setminus \{\text{-}\})^*$, and $L_2 = (\Sigma \setminus \{\text{-}\})^* b$. Then

$$\begin{array}{rcl}\mathsf{Pre}_{R_f}(L_1) \cap \mathsf{Pre}_{R_f}(L_2) & = & (a(\Sigma \setminus \{\text{-}\})^* \cup a(\Sigma \setminus \{\text{-}\})^*\text{-}\Sigma^* \cup \Sigma^*\text{-}a(\Sigma \setminus \{\text{-}\})^* \cup \Sigma^*\text{-}a(\Sigma \setminus \{\text{-}\})^*\text{-}\Sigma^*) \ \cap \\ & & ((\Sigma \setminus \{\text{-}\})^* b \cup (\Sigma \setminus \{\text{-}\})^* b\text{-}\Sigma^* \cup \Sigma^*\text{-}(\Sigma \setminus \{\text{-}\})^* b \cup \Sigma^*\text{-}(\Sigma \setminus \{\text{-}\})^* b\text{-}\Sigma^*),\end{array}$$

which is different from

$$\mathsf{Pre}_{R_f}(L_1 \cap L_2) = a(\Sigma \setminus \{\text{-}\})^* b \cup a(\Sigma \setminus \{\text{-}\})^* b\text{-}\Sigma^* \cup \Sigma^*\text{-}a(\Sigma \setminus \{\text{-}\})^* b \cup \Sigma^*\text{-}a(\Sigma \setminus \{\text{-}\})^* b\text{-}\Sigma^*.$$

For instance, $a\text{-}b \in \mathsf{Pre}_{R_f}(L_1) \cap \mathsf{Pre}_{R_f}(L_2)$, but $a\text{-}b \notin \mathsf{Pre}_{R_f}(L_1 \cap L_2)$. □

The distributivity of the pre-image of string functions means that, for each $y := f(\vec{x})$ and $y \in \mathcal{A}$ with $\mathcal{L}(\mathcal{A}) = \mathcal{L}(\mathcal{A}_1) \cap \cdots \cap \mathcal{L}(\mathcal{A}_s)$, we can compute $\mathsf{Pre}_{R_f}(\mathcal{A})$ by *separately* computing the pre-image $\mathsf{Pre}_{R_f}(\mathcal{A}_i)$ for each assertion $y \in \mathcal{A}_i$, i.e., no product construction is performed.

Moreover, to obtain the EXPSPACE upper bound, we need to carefully examine the complexity of the pre-image computation for each string operation. The pre-image computation of the replaceAll function has recently been addressed in [Chen et al. 2018a]. In the following, we tackle other string functions in SL[∘, replaceAll, reverse, FFT]. To this end, we utilise a succinct representation of the conjunction of a special class of regular languages, called *conjunctive FAs*.

Definition 5.3 (Conjunctive FA). A conjunctive FA is a pair $(\mathcal{A}, \Omega)$, where $\mathcal{A} = (Q, \delta)$ is a transition graph and $\Omega \subseteq Q \times Q$ is called a *conjunctive acceptance condition*. The language defined by $(\mathcal{A}, \Omega)$, denoted by $\mathcal{L}(\mathcal{A}, \Omega)$, is $\bigcap\limits_{(q,q') \in Q} \mathcal{L}((Q, q, \{q'\}, \delta))$. The size of $(\mathcal{A}, \Omega)$, is defined as $|Q|$.

Note that the conjunctive FA $(\mathcal{A}, \Omega)$ is exponentially more succinct than the product automaton of all FAs $(Q, q, \{q'\}, \delta)$ for $(q, q') \in \Omega$. For a string operation $f : (\Sigma^*)^r \to \Sigma^*$, we use $\mathsf{Pre}_{R_f}(\mathcal{A}, \Omega)$ to denote the pre-image of $\mathcal{L}(\mathcal{A}, \Omega)$ under R_f. A *conjunctive representation* of $\mathsf{Pre}_{R_f}(\mathcal{A}, \Omega)$ is a collection $((\mathcal{A}_j^{(1)}, \Omega_j^{(1)}), \ldots, (\mathcal{A}_j^{(r)}, \Omega_j^{(r)}))_{1 \le j \le n}$, where each atom $(\mathcal{A}_j^{(i)}, \Omega_j^{(i)})$ is a conjunctive FA.

Based on conjunctive FAs and the fact that the product construction of regular constraints can be avoided, we show the EXPSPACE upper bound for SL[∘, replaceAll, reverse, FFT].

PROPOSITION 5.4. *The path feasibility of* SL[∘, replaceAll, reverse, *FFT] is in* EXPSPACE.

The proof of Proposition 5.4 can be found in the full version of this article [Chen et al. 2018b]. For the EXPSPACE lower bound, it has been shown in [Lin and Barceló 2016] that SL with *FFT* and ∘ is EXPSPACE-hard. (Note that all transducers used in the reduction therein are functional.) To complement this result, we show that SL with replaceAll solely is already EXPSPACE-hard. This result is interesting in its own right. In particular, it solves an open problem left over in [Chen et al. 2018a].

PROPOSITION 5.5. *The path feasibility problem for* SL[replaceAll] *is* EXPSPACE*-hard.*

The proof of the above proposition is by a reduction from a tiling problem over an exponentially wide corridor (see Section 4.2 for a definition). We give the full proof in the full version of this article [Chen et al. 2018b].

5.2 The fragment SL[∘, sreplaceAll, reverse, FT]

Theorem 5.1 shows that in SL, if the transducers are restricted to be functional and one-way, then the complexity of the path feasibility problem becomes EXPSPACE-complete. In the following, we show that the same complexity bound holds if the $\mathsf{replaceAll}_e$ function is made unary, whereas the transducers are allowed to be relational. We remark that the proof of the complexity upper-bound deviates from that of SL[∘, replaceAll, reverse, FFT].

THEOREM 5.6. *The path feasibility of* SL[∘, sreplaceAll, reverse, *FT] is* EXPSPACE*-complete.*

The lower-bound in Theorem 5.6 follows from that in [Lin and Barceló 2016] for ∘ and FTs. We focus on the upper-bound. Let S be a symbolic execution in SL[∘, sreplaceAll, reverse, FT]. Without loss of generality, we assume that the subject parameters of the sreplaceAll functions in S are always string variables (otherwise it can be eliminated). We have the four-step procedure below:

(1) At first, for each assertion $\mathbf{assert}(R(\vec{x}))$ in S, we nondeterministically choose one disjunct $(\mathcal{A}_1, \ldots, \mathcal{A}_r)$ of the representation of R, replace $\mathbf{assert}(R(\vec{x}))$ with the sequence of assertions $\mathbf{assert}(x_1 \in \mathcal{A}_1); \ldots; \mathbf{assert}(x_r \in \mathcal{A}_r)$. Let S_1 be the resulting program. Note that the size of S_1 is linear in that of S.

(2) Transform each assignment of the form $y := \mathsf{replaceAll}_e(x, u)$ in S_1 with e a regular expression and u a string constant, into $y := T_{e,u}(x)$, where $T_{e,u}$ is an FT corresponding to $\mathsf{replaceAll}_e(\cdot, u)$ that can be constructed from e, u in exponential time ([Chen et al. 2018a]). Let S_2 denote the resulting program.

(3) Remove all the occurrences of the $\circ$ operator from S_2, resulting in S_3. This step can be done in nondeterministic exponential time w.r.t. the size of S_1 (not S_2), thus the size of S_3 is at most exponential in that of S_1.

(4) Finally, reduce S_3 into a program S_4 that contains no occurrences of the reverse function. The reduction is done in polynomial time.

Note that S_4 is a program that contains only FTs and assertions of the form $y \in \mathcal{A}$ the size of which is exponential in that of S. According to [Barceló et al. 2013][Theorem 6.7], the path feasibility of a symbolic execution that contains only FTs and assertions of the form $y \in \mathcal{A}$ can be solved in polynomial space. Therefore, we conclude that the path feasibility of SL[$\circ$, sreplaceAll, reverse, FT] is in nondeterministic exponential space, thus in EXPSPACE by Savitch's theorem.

Since the first step is clear and the second step was shown in [Chen et al. 2018a], we will focus on the third and fourth step.

The third step is similar to the proof of Theorem 5 in [Lin and Barceló 2016]: The main idea is to do a bottom-up induction on the structure of the dependency graph (namely starting from the input variables) and split each variable into multiple segments represented by fresh variables. (In the current setting, one additional inductive case should be added to deal with the reverse function.) Crucially the number of fresh variables introduced in the third step depends only on the structure of the dependency graph, but is independent of the size of the transducers or automata in S_2. Therefore, there are at most exponentially (in the size of S_1) many fresh variables are introduced. The transducers or automata in S_3 are obtained from those of S_2 by designating one initial and one final state respectively. Therefore, the size of S_3 is at most exponential in that of S_1.

We mainly describe the fourth step, i.e., to eliminate the reverse function while preserving path feasibility. During this course, we need to introduce *reversed transducers*. For an FT T, we define FT T^π by reversing the direction of each transition of T and swapping initial and final states.

Let X denote the set of variables occurring in S_3. The fourth step comprises the following substeps.

(4.1) For each $x \in X$, add a new variable $x^{(\pi)}$, which intuitively denotes the reverse of x.

(4.2) Remove each $y := \mathsf{reverse}(x)$ in S_3 and replace each occurrence of y in S_3 with $x^{(\pi)}$.

(4.3) In this substep, we intend to make sure for each variable $x \in X$, it cannot be the case that both x and $x^{(\pi)}$ occur. To this end, we sequentially process the statements of S_3, as follows. Mark all the remaining assignments as unprocessed. Repeat the following procedure until all the assignments are processed:

- If the first unprocessed statement is of the form $y := T(x^{(\pi)})$ (resp. $y^{(\pi)} = T(x^{(\pi)})$) and x occurs in some processed assignment, then replace $y := T(x^{(\pi)})$ (resp. $y^{(\pi)} := T(x^{(\pi)})$) by $y^{(\pi)} := T^\pi(x)$ (resp. $y := T(x)$) and mark the new statement as processed.
- If the first unprocessed statement is of the form $y^{(\pi)} := T(x)$ (resp. $y := T(x)$) and $x^{(\pi)}$ occurs in some processed assignment, then replace $y^{(\pi)} := T(x)$ (resp. $y := T(x)$) by $y := T^\pi(x^{(\pi)})$ (resp. $y^{(\pi)} := T^\pi(x^{(\pi)})$) and mark the new statement as processed.
- For all the other cases, mark the first unprocessed statement as processed.

By induction, we can show that in each step of the above procedure, for each variable $x \in X$, at most one of x or $x^{(\pi)}$ occurs in the processed assignments. We then have that, after the step (**4.3**), for each $x \in X$, x and $x^{(\pi)}$ can *not* both occur in the assignments.

(4.4) For each variable $x \in X$, if x occurs in the assignments, then replace each regular constraint of the form $x^{(\pi)} \in \mathcal{L}(\mathcal{A})$ by $x \in \mathcal{L}(\mathcal{A}^{\pi})$, otherwise, replace each regular constraint of the form $x \in \mathcal{L}(\mathcal{A})$ by $x^{(\pi)} \in \mathcal{L}(\mathcal{A}^{\pi})$.

Recall that we use S_4 to denote the symbolic execution that results from executing the fourth step on S_3. From the arguments above, we know that for each $x \in X$, exactly one of x or $x^{(\pi)}$ occurs in S_4, but not both. Therefore, S_4 is a symbolic execution in SL[$\circ$, sreplaceAll, reverse, FT] that contains only FTs and regular constraints.

The following example illustrates the fourth step.

Example 5.7. Consider the symbolic execution

$$z := T(x);\ y := \mathsf{reverse}(x);\ z' := T'(y);\ \textbf{assert}(x \in \mathcal{A}_1);\ \textbf{assert}(z \in \mathcal{A}_2);\ \textbf{assert}(z' \in \mathcal{A}_3)$$

where T, T' are FTs and $\mathcal{A}_1, \mathcal{A}_2, \mathcal{A}_3$ are FAs. In the first substep, we add the variables $x^{(\pi)}, y^{(\pi)}, z^{(\pi)}, (z')^{(\pi)}$. In the second substep, we remove $y := \mathsf{reverse}(x)$ and replace y with $x^{(\pi)}$, resulting in the program

$$z := T(x);\ z' := T'(x^{(\pi)});\ \textbf{assert}(x \in \mathcal{A}_1);\ \textbf{assert}(z \in \mathcal{A}_2);\ \textbf{assert}(z' \in \mathcal{A}_3).$$

In the third substep, since when processing $z' := T'(x^{(\pi)})$, x has already occurred in the processed assignment $z := T(x)$, we transform $z' := T'(x^{(\pi)})$ into $(z')^{(\pi)} := T'(x)$, resulting in the program

$$z := T(x);\ (z')^{(\pi)} := T'(x);\ \textbf{assert}(x \in \mathcal{A}_1);\ \textbf{assert}(z \in \mathcal{A}_2);\ \textbf{assert}(z' \in \mathcal{A}_3).$$

Note that after the third substep, $x, z, (z')^{(\pi)}$ occur in the assignments, but none of $x^{(\pi)}, z^{(\pi)}$ and z' do. In the fourth substep, we replace $\textbf{assert}(z' \in \mathcal{A}_3)$ with $\textbf{assert}((z')^{(\pi)} \in \mathcal{A}_3^{\pi})$ and get the following symbolic execution in SL[$\circ$, sreplaceAll, reverse, FT] where only FTs and regular constraints occur,

$$z := T(x);\ (z')^{(\pi)} := T'(x);\ \textbf{assert}(x \in \mathcal{A}_1);\ \textbf{assert}(z \in \mathcal{A}_2);\ \textbf{assert}((z')^{(\pi)} \in \mathcal{A}_3^{\pi}).$$

□

As mentioned in Section 1, our algorithm reduces the problem to constraints which can be handled by the existing solver SLOTH [Holík et al. 2018].

Extensions with length constraints. Apart from regular constraints in the assertion, length constraints are another class of commonly used constraints in string manipulating programs. Some simple length constraints can be encoded by regular constraints, as partially exemplified in Example 3.2. Here, we show that when SL[$\circ$, sreplaceAll, reverse, FT] is extended with (much more) general length constraints, the EXPSPACE upper bound can still be preserved. We remark that, in contrast, if SL[$\circ$, replaceAll, reverse, FFT] is extended with length constraints, then the path feasibility problem becomes undecidable, which has already been shown in [Chen et al. 2018a].

To specify length constraints properly, we need to slightly extend our constraint language. In particular, we consider variables of type Int, which are usually referred to as *integer variables* and range 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.

Definition 5.8 (Length assertions). A length assertion is of the form $\textbf{assert}(a_1 t_1 + \cdots + a_n t_n \leq d)$, where $a_1, \ldots, a_n, d \in \mathbb{Z}$ are integer constants (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.

Theorem 5.9. *The path feasibility of* SL[$\circ$, sreplaceAll, reverse*, FT] extended with length assertions is* EXPSPACE*-complete.*

The proof of the theorem is given in the full version of this article [Chen et al. 2018b]. Other related constraints, such as character assertions and IndexOf assertions, can be encoded by length assertions defined in Definition 5.8 together with regular constraints. Since they are not the focus of this paper, we omit the details here.

We observe that Theorem 5.9 suggests that the two semantic conditions (roughly speaking, being a recognisable relation) is only a sufficient, but not necessary, condition of decidability of path feasibility. This is because length assertions can easily deviate from recognisable relations (for instance $|x_1| = |x_2|$ does not induce a recognisable relation over Σ^*), but decidability still remains.

6 IMPLEMENTATION

We have implemented our decision procedure for path feasibility in a new tool OSTRICH, which is built on top of the SMT solver Princess [Rümmer 2008]. OSTRICH implements the optimised decision procedure for string functions as described in Section 5.1 (i.e. using distributivity of regular constraints across pre-images of functions) and has built-in support for concatenation, reverse, FFT, and replaceAll. Moreover, since the optimisation only requires that string operations are functional, we can also support additional functions that satisfy **RegInvRel**, such as replace$_e$ which replaces only the first (leftmost and longest) match of a regular expression. OSTRICH is extensible and new string functions can be added quite simply (Section 6.3).

Our implementation adds a new theory solver for conjunctive formulas representing path feasibility problems to Princess (Section 6.1). This means that we support disjunction as well as conjunction in formulas, as long as every conjunction of literals fed to the theory solver corresponds to a path feasibility problem. OSTRICH also implements a number of optimisations to efficiently compute pre-images of relevant functions (Section 6.2). OSTRICH is entirely written in Scala and is open-source. We report on our experiments with OSTRICH in Section 6.4. The tool is available on GitHub[5]. The artifact is available on the ACM DL.

6.1 Depth-First Path Feasibility

We first discuss the overall decision procedure for path feasibility implemented in OSTRICH. The procedure performs depth-first exploration of the search tree resulting from repeatedly splitting the disjunctions (or unions) in the recognisable pre-images of functions. Similar to the DPLL/CDCL architecture used in SAT solvers, our procedure computes conflict sets and applies back-jumping [Nieuwenhuis et al. 2004] to skip irrelevant parts of the search tree.

For solving, a path feasibility problem is represented as a set *funApps* of assignments $x := f(\bar{y})$, and a set *regex* of regular expression constraints $x \in L$, with x being a string variable and $\bar{y}$ a vector of string variables. The set *funApps* by definition contains at most one assignment for each variable x, and by nature of a path there are no cyclic dependencies. We make two further simplifying assumptions: (i) the set *regex* is on its own satisfiable, i.e., the constraints given for each variable x are consistent; and (ii) for each variable that occurs as the left-hand side of an assignment $x := f(\bar{y})$, there is at least one constraint $x \in L$ in the set *regex*. Assumption (i) boils down to checking the non-emptiness of intersections of regular languages, i.e., to a reachability check in the product

[5] https://github.com/pruemmer/ostrich

Algorithm 1: Recursive function *findModel* defining depth-first model construction for SL

```
Input: Sets active, passive of regex constraints $x \in L$; acyclic set funApps of assignments $x := f(\bar{y})$.
Result: Either Model($m$) with $m$ a model satisfying all constraints and function applications;
        or Conflict($s$) with $s$ a set of regex constraints that is inconsistent with funApps.
begin
    if $active = \emptyset$ then
        /* Extract a model by solving constraints and evaluating functions */
        $leafTerms \leftarrow \{x \mid x$ occurs in $passive \cup funApps\} \setminus \{x \mid (x := f(\bar{y})) \in funApps\}$;
        $leafModel \leftarrow \{x \mapsto w \mid x \in leafTerms,\ w$ satisfies all constraints on $x$ in $passive\}$;
        $m \leftarrow \mu m .\ leafModel \cup \{x \mapsto f(\bar{w}) \mid (x := f(\bar{y})) \in funApps,\ m(\bar{y}) = \bar{w}$ is defined$\}$;
        return Model($m$)
    else
        /* Compute the pre-image of one of the active constraints */
        choose a constraint $x \in L$ in active;
        if there is an assignment $x := f(y_1, \ldots, y_r)$ defining $x$ in funApps then
            $cset \leftarrow \{x \in L\}$ ;    /* start constructing a conflict set */
            compute the pre-image $f^{-1}(L) = \bigcup_{i=1}^{n} L_1^{(i)} \times \cdots \times L_r^{(i)}$;
            $act \leftarrow active \setminus \{x \in L\},\ pas \leftarrow passive \cup \{x \in L\}$;
            for $i \leftarrow 1$ to $n$ do
                $newRegexes \leftarrow \{y_1 \in L_1^{(i)}, \ldots, y_r \in L_r^{(i)}\} \setminus (act \cup pas)$;
                if $act \cup pas \cup newRegexes$ is inconsistent then
                    compute an unsatisfiable core $c \subseteq act \cup pas \cup newRegexes$;
                    $cset \leftarrow cset \cup (c \setminus \{y_1 \in L_1^{(i)}, \ldots, y_r \in L_r^{(i)}\})$;
                else
                    switch findModel($act \cup newRegexes,\ pas,\ funApps$) do
                        case Model($m$)
                            return Model($m$);
                        case Conflict($s$)
                            if $s \cap newRegexes = \emptyset$ then
                                return Conflict($s$) ;    /* back-jump */
                            else
                                $cset \leftarrow cset \cup (s \setminus \{y_1 \in L_1^{(i)}, \ldots, y_r \in L_r^{(i)}\})$;
            return Conflict($cset$) ;    /* backtrack */
        else
            return findModel($active \setminus \{x \in L\},\ passive \cup \{x \in L\},\ funApps$);
```

transition system, and can be done efficiently in practical cases. Both assumptions could easily be relaxed, at the cost of making the algorithm slightly more involved.

6.1.1 The exploration function. The algorithm is presented as a recursive function *findModel* in pseudo-code in Algorithm 1. The function maintains two sets *active*, *passive* of regular expression constraints, kept as parameters, and in addition receives the set *funApps* as parameter. *Active* regular expression constraints are those for which no pre-images have been computed yet, while *passive* are the constraints that have already gone through pre-image computation. In the initial call *findModel*(*regex*, $\emptyset$, *funApps*) of the function, *active* is chosen to be *regex*, while *passive* is empty.

Depending on the status of a path feasibility problem, *findModel* can produce two outcomes:

- *Model*(m), where m is maps string variables to words that satisfy the regular expression constraints in *active* $\cup$ *passive* and the assignments in *funApps*, interpreted as equations.
- *Conflict*(s), with $s \subseteq$ *active* $\cup$ *passive* being a *conflict set*, i.e., a set of constraints that is inconsistent with *funApps*. The set $s \cup$ *funApps* can be interpreted as an unsatisfiable core of the path feasibility problem, and is used to identify irrelevant splits during the search.

6.1.2 Implementation in detail. Model construction terminates with a positive result when the set *active* becomes empty (line 2), in which case it is only necessary to compute a model m (lines 3–6). For this, the algorithm first computes the set of variables that do not occur on the left-hand side of any assignment (line 3). The values of such leaf terms can be chosen arbitrarily, as long as all derived regular expression constraints are satisfied (line 4). The values of all other variables are extracted from the assignments in *funApps*: whenever an assignment $x := f(\bar{y})$ is found for which all argument variables already have a value, also the value of the left-hand side x is known (line 5).

Otherwise, one of the active constraints $x \in L$ is selected for pre-image computation (line 8). For the correctness it is irrelevant in which order the constraints are selected, and branching heuristics from the SAT world might be applicable. Our current implementation selects the constraints in the fixed order in which the constrained variables occur on the path, starting with constraints at the end of the path. If x is a left-hand side of an assignment, the pre-image of L is a recognisable relation that can be represented through regular languages (line 11); the constraint $x \in L$ then becomes passive in subsequent recursive calls (line 12).

The algorithm then iterates over the disjuncts of the pre-image (line 13), generates new regular expression constraints for the function arguments (line 14), and checks whether any disjuncts lead to a solution. During the iteration over disjuncts, the algorithm builds up a conflict set *cset* collecting constraints responsible for absence of a solution (lines 10, 17, 26). If the new constraints are inconsistent with existing constraints (line 15), the disjunct does not have to be considered further; the algorithm then computes a (possibly minimal) unsatisfiable subset of the constraints, and adds it to the conflict set *cset*. Otherwise, *findModel* is called recursively (line 19). If the recursive call produces a model, no further search is necessary, and the function returns (lines 20-21). Similarly, if the recursive call reports a conflict that is independent of the generated regular expression constraints, it follows that no solution can exist for any of the disjuncts of the pre-image, and the function can return immediately (lines 23–24). In case of other conflicts, the conflict set *cset* is extended (line 26), and finally returned to explain why no model could be found (line 27).

6.2 Optimisation of Pre-Image Computation

We have optimised the pre-image computation of the concatenation and replaceAll operations.

6.2.1 Concatenation. The pre-image of a regular language L for concatenation $x := y \circ z$ can be computed by representing L as an FA, say with n states, and generating a union $\bigcup_{i=1}^{n} L_1^{(i)} \times L_2^{(i)}$ in which the accepting state of $L_1^{(i)}$ and the initial state of $L_2^{(i)}$ iterate over all n states of L [Abdulla et al. 2014]. In practice, most of the resulting n cases are immediately inconsistent with other regular expression constraints in *active* $\cup$ *passive*; this happens for instance when the length of y or z are already predetermined by other constraints. Our implementation therefore filters the considered languages $L_1^{(i)}, L_2^{(i)}$ upfront using length information extracted from other constraints.

6.2.2 The replaceAll *Function.* We implement $\mathsf{replaceAll}_e$ by reduction to $\mathsf{replaceAll}_a$ for a single character a. We translate all $x := \mathsf{replaceAll}_e(y, z)$ into $y' := T_e(y); x := \mathsf{replaceAll}_\flat(y', z)$ where $\flat$ is a special character disjoint from the rest of the alphabet. The transducer T_e copies the contents of y and replaces all left-most and longest subwords satisfying the regular expression e with $\flat$. This construction uses a *parsing automaton* and details can be found in [Chen et al. 2018a]. We first recall how $\mathsf{replaceAll}_a$ can be tackled, before explaining the inefficiencies and the solution we use.

Naive Recognisability. Suppose we have an FA $\mathcal{A}_x$ giving a regular constraint on x. We need to translate this automaton into a sequence of regular constraints on y and z. To do this, we observe that all satisfying assignments to x must take the form $u_1 w_z u_2 w_z \ldots w_z u_n$ where w_z is the value of z, and $u_1 a u_2 a \ldots a u_n$ is the value of y. Moreover, each word u_i cannot contain the character a since it would have been replaced by w_z. The (satisfying) assignment to x must be accepted by $\mathcal{A}_x$. Thus, we can extract from an accepting run of $\mathcal{A}_x$ a set of pairs of states Q_z, which is the set of all pairs (q, q') such that the run of $\mathcal{A}_x$ moves from q to q' while reading a copy of w_z. Then, we can obtain a new FA $\mathcal{A}_y$ by removing all a-transitions from $\mathcal{A}_x$ and then adding a-transitions (q, a, q') for each $(q, q') \in Q_z$. It is easy to verify that there is an accepting run of $\mathcal{A}_y$ over $u_1 a u_2 a \ldots a u_n$. Similarly, we define $\mathcal{A}_z$ to be the intersection of $\mathcal{A}_x(q, \{q'\})$ for all $(q, q') \in Q_z$. We know by design that there is an accepting run of $\mathcal{A}_z$ over w_z.

The value of Q_z above was extracted from an accepting run of $\mathcal{A}_x$. There are, of course, many possible accepting runs of $\mathcal{A}_x$, each leading to a different value of Q_z, and thus a different $\mathcal{A}_y$ and $\mathcal{A}_z$. Since each Q_z a set of pairs of states of $\mathcal{A}_x$, there are only a finite number of values that can be taken by Q_z. Thus, we can show the pre-image of $x := \mathsf{replaceAll}_a(y, a, z)$ is recognisable by enumerating all possible values of Q_z. For each Q_z we can produce a pair of automata $(\mathcal{A}_y^{Q_z}, \mathcal{A}_z^{Q_z})$ as described above. Thus, the pre-image can be expressed by $\bigcup_{Q_z} \left(\mathcal{L}\left(\mathcal{A}_x^{Q_z}\right) \times \mathcal{L}\left(\mathcal{A}_y^{Q_z}\right)\right)$.

Optimised Recognisability. A problem with the above algorithm is that there are an exponential number of possible values of Q_z. For example, if $\mathcal{A}_x$ has 10 states, there are 2^{100} possible values of Q_z; it is infeasible to enumerate them all. To reduce the number of considered pairs, we use the notion of a *Cayley Graph* [Dénes 1967; Zelinka 1981]. Note, this technique was already used by Chen [Chen 2018a,b] as part of an implementation of [Chen et al. 2018a].

Given an automaton $\mathcal{A}$ and a word w, we define

$$(\!|w|\!) = \{(q_0, q_n) \mid \text{there exists a run } q_0, \ldots, q_n \text{ of } \mathcal{A} \text{ over } w\}\ .$$

The number of distinct $(\!|w|\!)$ is finite for a given FA. We define Cayley Graphs in the context of FA.

Definition 6.1 (Cayley Graph). Given an FA $\mathcal{A} = (\Sigma, Q, q_0, F, \delta)$ the *Cayley Graph* of $\mathcal{A}$ is a graph (V, E) with nodes $V = \{(\!|w|\!) \mid w \in \Sigma^*\}$ and edges $E = \{((\!|w|\!), a, (\!|wa|\!)) \mid w \in \Sigma^* \wedge a \in \Sigma\}$.

For a given automaton, it is straight-forward to compute the Cayley Graph using a fixed point iteration: begin with $(\!|\epsilon|\!) \in V$, then, until a fixed point is reached, take some $(\!|w|\!)$ in V and add $(\!|wa|\!)$ for all $a \in \Sigma$. Note $(\!|wa|\!)$ is a simple composition of $(\!|w|\!)$ and $(\!|a|\!) = \{(q, q') \mid (q, a, q') \text{ is an edge of } \mathcal{A}\}$.

We claim that instead of enumerating all Q_z, one only needs to consider all $(\!|w|\!) \in V$. Since $(\!|w|\!)$ can be a value of Q_z, the restriction does not increase the set of potential solutions. We need to argue that it does not reduce them. Hence, take some satisfying value $u_1 w_z u_2 w_z \ldots w_z u_n$ of x. One can easily verify that w_z is accepted by the $\mathcal{A}_z$ constructed from $(\!|w_z|\!)$. Moreover, $u_1 a u_2 a \ldots a u_n$ is accepted by the corresponding $\mathcal{A}_y$. Thus we have not restricted the algorithm.

From experience, it is reasonable to hope that the Cayley Graph has far fewer nodes than the set of all potential Q_z. Moreover, we can further improve the enumeration by considering any other regular constraints $\mathcal{A}_z^1, \ldots, \mathcal{A}_z^m$ that may exist on the value of z. As a simple example, if we had $\mathbf{assert}(z \in b^*); x := \mathsf{replaceAll}_a(y, z); \mathbf{assert}(x \in \mathcal{A}_x)$ where $\mathcal{A}_x$ has initial state q_0 and accepting states q_1 and q_2 with transitions (q_0, a, q_1) and (q_0, b, q_1), there is no need to consider $(\!|a|\!) = \{(q_0, q_1)\}$ since z cannot take the value a without violating $\mathbf{assert}(z \in b^*)$.

Using this observation, assume we already know that z's value must be accepted by $\mathcal{A}_z^1, \ldots, \mathcal{A}_z^m$. Instead of building the Cayley Graph alone, we build a product of the Cayley Graph and $\mathcal{A}_z^1, \ldots, \mathcal{A}_z^m$

on the fly. This product has states $(\llparenthesis w \rrparenthesis, q_1, \ldots, q_m)$ for some $w \in \Sigma^*$ and $q_1, \ldots, q_m$ states of $\mathcal{A}_z^1, \ldots, \mathcal{A}_z^m$ respectively. The only $\llparenthesis w \rrparenthesis$ we need to consider are those such that $(\llparenthesis w \rrparenthesis, q_1, \ldots, q_m)$ is a (reachable) product state, and, moreover, $q_1, \ldots, q_m$ are accepting states of $\mathcal{A}_z^1, \ldots, \mathcal{A}_z^m$.

This technique first speeds up the construction of the Cayley Graph by limiting which nodes are generated, and second, avoids values of Q_z which are guaranteed to be unsatisfying.

6.3 Extensibility

One may extend OSTRICH to include any string function with a recognisable pre-image *without having to worry about other parts of the solver.* We give an example of adding a new reverse function.

We have defined a Scala trait `PreOp`. To add a string function, one defines a new Scala object with this trait. This requires two methods described below. An example object is given in Figure 3.

The first method is `eval`, which implements the string function. It takes a sequence of strings represented as sequence of integers. For reverse, this method reverses the argument. For $\mathsf{replaceAll}_a(x, y)$, the `eval` function would take a sequence of two arguments (the values of x and y respectively) and replace all a characters in x with the value of y to produce the result. The return value can be `None` if the function is not applicable to the given arguments.

The second method `apply` performs the pre-image computation. It takes two arguments: a sequence of constraints on the arguments (`argumentConstraints`), and a constraint on the result (`resultConstraint`). The result constraint is represented as an `Automaton` that accepts the language for which we are computing f^{-1}. The argument constraints are represented as sequences of sequences of automata: for each argument of the function there will be one sequence of automata. These constraints give further information on what is known about the constraints on the arguments of the function. For example, if we are computing $x := \mathsf{reverse}(y)$ and elsewhere we have determined y must be accepted by both $\mathcal{A}_1$ and $\mathcal{A}_2$, then the first (and only) element of the argument constraints will be the sequence $\mathcal{A}_1, \mathcal{A}_2$. It is not necessary to use these constraints, but they may help to optimise the pre-image computation (as in the case of replaceAll described above).

The return value of `apply` is a pair. The first element is the pre-image (a recognisable relation). It is represented as an iterator over sequences of automata, where each sequence corresponds to a tuple $(\mathcal{A}_1, \ldots, \mathcal{A}_n)$ of the relation. The second element is a list of the argument constraints used during the pre-image computation; this information is needed to compute correct conflict sets in Algorithm 1. If the argument constraints were not used to optimise the pre-image computation, this value can be an empty list. If the arguments were used, then those constraints which were used should be returned in the same format as the argument constraints.

For convenience, assume that we have already implemented a reversal operation on automata as `AutomataUtils.reverse`. A `PreOp` object for the reverse function is given in Figure 3. The `apply` method reverses the result constraint, and returns an iterator over this single automaton. The second component of the return value is an empty list since the argument constraints were not used. The `eval` method simply reverses its first (and only) argument.

To complete the addition of the function, the reverse function is registered in the OSTRICH string theory object. The function is then ready to be used in an SMT-LIB problem, e.g. by writing the assertion `(assert (= x (user_reverse y)))`.

6.4 Experiments

We have compared OSTRICH with a number of existing solvers on a wide range of benchmarks. In particular, we compared OSTRICH with CVC4 1.6 [Liang et al. 2014], SLOTH [Holík et al. 2018],

```
object ReversePreOp extends PreOp {
  def apply(argumentConstraints : Seq[Seq[Automaton]], resultConstraint : Automaton)
          : (Iterator[Seq[Automaton]], Seq[Seq[Automaton]]) = {
    val revAut = AutomataUtils.reverse(resultConstraint)
    (Iterator(Seq(revAut)), List())
  }
  def eval(arguments : Seq[Seq[Int]]) : Option[Seq[Int]] = Some(arguments(0).reverse)
}
```

Fig. 3. A PreOp for the reverse function.

and Z3[6] configured to use the Z3-str3 string solver [Berzish et al. 2017]. We considered several sets of benchmarks which are described in the next sub-section. The results are given in Section 6.4.2.

In [Holík et al. 2018] SLOTH was compared with S3P [Trinh et al. 2016] where inconsistent behaviour was reported. We contacted the S3P authors who report that the current code is unsupported; moreover, S3P is being integrated with Z3. Hence, we do not compare with this tool.

6.4.1 Benchmarks. The first set of benchmarks we call Transducer. It combines the benchmark sets of Stranger [Yu et al. 2010] and the mutation XSS benchmarks of [Lin and Barceló 2016]. The first (sub-)set appeared in [Holík et al. 2018] and contains instances manually derived from PHP programs taken from the website of Stranger [Yu et al. 2010]. It contains 10 formulae (5 sat, 5 unsat) each testing for the absence of the vulnerability pattern `.*<script.*` in the program output. These formulae contain between 7 and 42 variables, with an average of 21. The number of atomic constraints ranges between 7 and 38 and averages 18. These examples use disjunction, conjunction, regular constraints, and concatenation, replaceAll. They also contain several one-way functional transducers (defined in SMTLIB in [Holík et al. 2018]) encoding functions such as `addslashes` and `trim` used by the programs. Note that transducers have been known for some time to be a good framework for specifying sanitisers and browser transductions (e.g., see the works by Minamide, Veanes, Saxena, and others [D'Antoni and Veanes 2013; Hooimeijer et al. 2011; Minamide 2005; Weinberger et al. 2011]), and a library of transducer specifications for such functions is available (e.g. see the language BEK [Hooimeijer et al. 2011]).

The second (sub-)set was used by [Holík et al. 2018; Lin and Barceló 2016] and consists of 8 formulae taken from [Kern 2014; Lin and Barceló 2016]. These examples explore mutation XSS vulnerabilities in JavaScript programs. They contain between 9 and 12 variables, averaging 9.75, and 9-13 atomic constraints, with an average of 10.5. They use conjunctions, regular constraints, concatenation, replaceAll, and transducers providing functions such as `htmlEscape` and `escapeString`.

Our next set of benchmarks, SLOG, came from the SLOG tool [Wang et al. 2016] and consist of 3,392 instances. They are derived from the security analysis of real web applications and contain 1-211 string variables (average 6.5) and 1-182 atomic formula (average 5.8). We split these benchmarks into two sets SLOG (replace) and SLOG (replaceall). Each use conjunction, disjunction, regular constraints, and concatenation. The set SLOG (replace) contains 3,391 instances and uses replace. SLOG (replaceall) contains 120 instances using the replaceAll operation.

Our next set of benchmarks Kaluza is the well-known set of *Kaluza* benchmarks [Saxena et al. 2010] restricted to those instances which satisfy our semantic conditions (roughly ~80% of the benchmarks). Kaluza contains concatenation, regular constraints, and length constraints, most of which admit regular monadic decomposition. There are 37,090 such benchmarks (28 032 sat).

[6] Github version 2aeb814f4e7ae8136ca5aeeae4d939a0828794c8

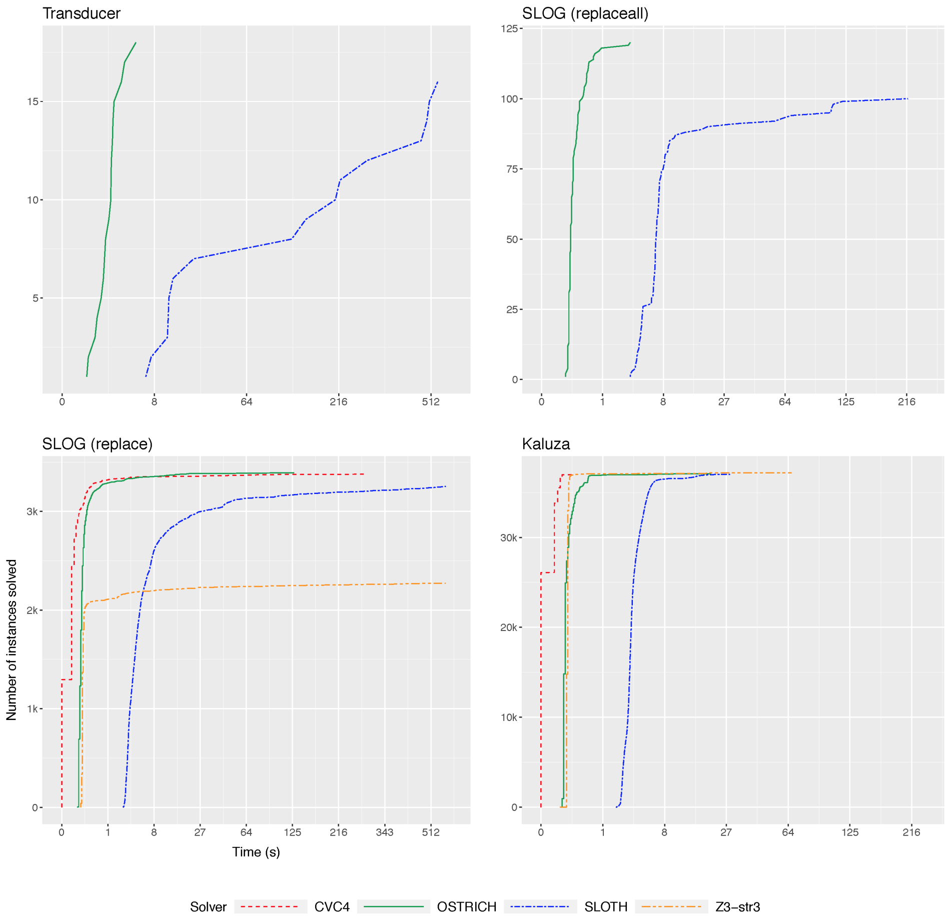

Fig. 4. Comparison of solvers on several sets of benchmarks.

We also considered the benchmark set of [Chen 2018a,b]. This contains 42 hand-crafted benchmarks using regular constraints, concatenation, and replaceAll with variables in both argument positions. The benchmarks contain 3-7 string variables and 3-9 atomic constraints.

6.4.2 Results. We compared the tools on an AMD Opteron 2220 SE machine, running 64-bit Linux and Java 1.8, with the heap space of each job limited to 2 GB. We used a timeout of 240s for each Kaluza problem, and 600s for the other benchmarks. Figure 4 summarises our findings as cactus plots. For each benchmark set, we plot the time in seconds on a cubic-root scale against the number of benchmarks solved (individually) within that time. The extent of each line on the Time axis gives the maximum time in seconds required to solve any instance in the set. When a solver is not plotted it is because it was unable to analyse the benchmark set.

For the Transducer set, OSTRICH solved all benchmarks taking a maximum of 4s. SLOTH did not answer 2 instances and was slower on the rest. This set is not supported by CVC4 or Z3-str3.

For the SLOG (replaceall) set, OSTRICH solved all 120 instances within a few seconds, while SLOTH only solved 100. CVC4 and Z3-str3 were omitted as they do not support replaceAll constraints.

For the SLOG (replace) set, OSTRICH was also able to solve all instances within the timeout. CVC4 was able to solve all but 13 of the benchmarks. Similarly, SLOTH was unable to solve 138 instances, while Z3-str3 could not solve 1,118. We note that Z3-str3 gave inconsistent results for 18 of these benchmarks, an issue that could not be conclusively resolved before submission..

For the Kaluza set, CVC4, Z3-str3, and OSTRICH were able to solve all instances. Since SLOTH does not support length constraints, it reported errors in 81 of these benchmarks. Otherwise all instances were solved within the timeout.

We were unable to install the tool of [Chen 2018b] for comparison with the benchmarks of [Chen 2018a,b]. Since OSTRICH was the only available tool supporting variables in both arguments of replaceAll, we do not provide a plot. We note that the most difficult benchmark took OSTRICH 1.56s to answer, with the second hardest requiring 0.34s.

Overall, CVC4 was the fastest for the constraints it was able to answer, while OSTRICH was the only solver which answered all benchmark instances. However, the runtime differences are fractions of a second. In terms of completeness guarantees and the type of string constraints supported, SLOTH is our nearest competitor, with the main difference being that OSTRICH supports variables in both argument positions of replaceAll, while SLOTH will only accept constant strings as the second argument. Our results show that OSTRICH outperforms SLOTH on all benchmark sets.

7 CONCLUSION

We proposed two general semantic conditions which together ensure the decidability of path feasibility with complex string operations including replaceAll, transducers, and concatenation. Our semantic conditions are satisfied by a wide range of complex string operations and subsume various existing string constraint languages (e.g. [Chen et al. 2018a; Lin and Barceló 2016]) and many current existing benchmarks (e.g. [Holík et al. 2018; Lin and Barceló 2016; Saxena et al. 2010; Wang et al. 2016; Yu et al. 2010]). Based on the semantic conditions, we developed a conceptually simple and generic decision procedure with an extensible architecture that allow a user to easily incorporate a user-defined function. After providing theoretical evidence via computational complexity that these semantic conditions might be too general to provide an efficient decision procedure, we proposed two different solutions. We advocated the first solution (prohibit nondeterminism in string operations) whenever possible since it permits a highly effective optimisation of the solver based on a kind of distributivity property of regular constraints across string functions. In fact, the extra restriction imposed by this is satisfied by most (but not all) existing benchmarking examples. We developed a new string solver OSTRICH that implements the first solution and demonstrate its efficacy against other competitive solvers on most existing benchmarks.

ACKNOWLEDGMENTS

We are grateful to the anonymous referees for their constructive and detailed comments. T. Chen is supported by the Engineering and Physical Sciences Research Council under Grant No. EP/P00430X/1 and the Australian Research Council under Grant No. DP160101652, DP180100691. 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). P. Rümmer is supported by the Swedish Research Council (VR) under grant 2014-5484, and by the Swedish Foundation for Strategic Research (SSF) under the project WebSec (Ref. RIT17-0011). Z. Wu is supported by the National Natural Science Foundation of China under Grant No. 61472474, 61572478, and 61872340, 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*. ACM, New York, NY, USA, 602–617. https://doi.org/10.1145/3062341.3062384

Parosh Aziz Abdulla, Mohamed Faouzi Atig, Yu-Fang Chen, Bui Phi Diep, Lukás Holík, Ahmed Rezine, and Philipp Rümmer. 2018. TRAU: SMT Solver for String Constraints. In *Formal Methods in Computer Aided Design, FMCAD 2018*. To appear.

Parosh Aziz Abdulla, Mohamed Faouzi Atig, Yu-Fang Chen, Lukás Holík, Ahmed Rezine, Philipp Rümmer, and Jari Stenman. 2014. String Constraints for Verification. In *Computer Aided Verification - 26th International Conference, CAV 2014*. Springer, 150–166. https://doi.org/10.1007/978-3-319-08867-9_10

Rajeev Alur and Jyotirmoy V. Deshmukh. 2011. Nondeterministic Streaming String Transducers. In *Automata, Languages and Programming - 38th International Colloquium, ICALP 2011, Zurich, Switzerland, July 4-8, 2011, Proceedings, Part II*. Springer, 1–20. https://doi.org/10.1007/978-3-642-22012-8_1

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

Pablo Barceló, Diego Figueira, and Leonid Libkin. 2013. Graph Logics with Rational Relations. *Logical Methods in Computer Science* 9, 3 (2013). https://doi.org/10.2168/LMCS-9(3:1)2013

Clark W. Barrett, Roberto Sebastiani, Sanjit A. Seshia, and Cesare Tinelli. 2009. Satisfiability Modulo Theories. In *Handbook of Satisfiability*. IOS Press, 825–885. https://doi.org/10.3233/978-1-58603-929-5-825

Michael Benedikt, Leonid Libkin, Thomas Schwentick, and Luc Segoufin. 2003. Definable relations and first-order query languages over strings. *J. ACM* 50, 5 (2003), 694–751. http://doi.acm.org/10.1145/876638.876642

Murphy Berzish, Vijay Ganesh, and Yunhui Zheng. 2017. Z3str3: A string solver with theory-aware heuristics. In *2017 Formal Methods in Computer Aided Design, FMCAD 2017, Vienna, Austria, October 2-6, 2017*. IEEE, 55–59. https://doi.org/10.23919/FMCAD.2017.8102241

Nikolaj Bjørner, Nikolai Tillmann, and Andrei Voronkov. 2009. Path feasibility analysis for string-manipulating programs. In *Tools and Algorithms for the Construction and Analysis of Systems, TACAS 2009*. Springer, 307–321. https://doi.org/10.1007/978-3-642-00768-2_27

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. https://doi.org/10.1002/malq.19880340410

Thierry Cachat and Igor Walukiewicz. 2007. The Complexity of Games on Higher Order Pushdown Automata. *CoRR* abs/0705.0262 (2007). arXiv:0705.0262 http://arxiv.org/abs/0705.0262

Cristian Cadar, Daniel Dunbar, and Dawson Engler. 2008. KLEE: Unassisted and Automatic Generation of High-coverage Tests for Complex Systems Programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation (OSDI'08)*. USENIX Association, Berkeley, CA, USA, 209–224. http://dl.acm.org/citation.cfm?id=1855741.1855756

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

Cristian Cadar and Koushik Sen. 2013. Symbolic Execution for Software Testing: Three Decades Later. *Commun. ACM* 56, 2 (Feb. 2013), 82–90. https://doi.org/10.1145/2408776.2408795

Olivier Carton, Christian Choffrut, and Serge Grigorieff. 2006. Decision problems among the main subfamilies of rational relations. *ITA* 40, 2 (2006), 255–275. https://doi.org/10.1051/ita:2006005

Taolue Chen, Yan Chen, Matthew Hague, Anthony W. Lin, and Zhilin Wu. 2018a. What is decidable about string constraints with the ReplaceAll function. *PACMPL* 2, POPL (2018), 3:1–3:29. https://doi.org/10.1145/3158091

Taolue Chen, Matthew Hague, Anthony W. Lin, Philipp Rümmer, and Zhilin Wu. 2018b. Decision Procedures for Path Feasibility of String-Manipulating Programs with Complex Operations. *CoRR* abs/1811.03167 (2018). arXiv:1811.03167 https://arxiv.org/abs/1811.03167

Yan Chen. 2018a. *Solving String Constraints with ReplaceAll Function*. Master's thesis. State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences, China.

Yan Chen. 2018b. Z3-replaceall. https://github.com/TinyYan/z3-replaceAll. Referred in Jan 2018.

Christian Choffrut. 2006. Relations over Words and Logic: A Chronology. *Bulletin of the EATCS* 89 (2006), 159–163.

Loris D'Antoni and Margus Veanes. 2013. Static Analysis of String Encoders and Decoders. In *Verification, Model Checking, and Abstract Interpretation, VMCAI*. Springer, 209–228. https://doi.org/10.1007/978-3-642-35873-9_14

Leonardo De Moura and Nikolaj Bjørner. 2011. Satisfiability modulo theories: introduction and applications. *Commun. ACM* 54, 9 (2011), 69–77. https://doi.org/10.1145/1995376.1995394

J. Dénes. 1967. Connections between transformation semigroups and graphs. In *Theory of Graphs*. Gordon & Breach.

Volker Diekert. 2002. Makanin's Algorithm. In *Algebraic Combinatorics on Words*, M. Lothaire (Ed.). Encyclopedia of Mathematics and its Applications, Vol. 90. Cambridge University Press, Chapter 12, 387–442. https://doi.org/10.1017/

CBO9781107326019.013

Joost Engelfriet and Hendrik Jan Hoogeboom. 2001. MSO definable string transductions and two-way finite-state transducers. *ACM Trans. Comput. Log.* 2, 2 (2001), 216–254. https://doi.org/10.1145/371316.371512

Emmanuel Filiot, Olivier Gauwin, Pierre-Alain Reynier, and Frédéric Servais. 2013. From Two-Way to One-Way Finite State Transducers. In *28th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2013, New Orleans, LA, USA, June 25-28, 2013.* IEEE Computer Society, 468–477. https://doi.org/10.1109/LICS.2013.53

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.* Springer, 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

Lukás Holík, Petr Janku, Anthony W. Lin, Philipp Rümmer, and Tomás Vojnar. 2018. String constraints with concatenation and transducers solved efficiently. *PACMPL* 2, POPL (2018), 4:1–4:32. https://doi.org/10.1145/3158092

Pieter Hooimeijer, Benjamin Livshits, David Molnar, Prateek Saxena, and Margus Veanes. 2011. Fast and Precise Sanitizer Analysis with BEK. In *20th USENIX Security Symposium, San Francisco, CA, USA, August 8-12, 2011, Proceedings.* USENIX Association. http://static.usenix.org/events/sec11/tech/full_papers/Hooimeijer.pdf

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

Artur Jez. 2016. Recompression: A Simple and Powerful Technique for Word Equations. *J. ACM* 63, 1 (2016), 4:1–4:51. https://doi.org/10.1145/2743014

Neil D. Jones. 2001. The expressive power of higher-order types or, life without CONS. *Journal of Functional Programming* 11, 1 (2001), 55–94. http://journals.cambridge.org/action/displayAbstract?aid=68581

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

Adam Kiezun, Vijay Ganesh, Shay Artzi, Philip J. Guo, Pieter Hooimeijer, and Michael D. Ernst. 2012. HAMPI: A solver for word equations over strings, regular expressions, and context-free grammars. *ACM Trans. Softw. Eng. Methodol.* 21, 4 (2012), 25. https://doi.org/10.1145/2377656.2377662

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

Dexter Kozen. 1997. *Automata and Computability.* Springer.

Daniel Kroening and Ofer Strichman. 2008. *Decision Procedures.* Springer.

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 *Computer Aided Verification - 26th International Conference, CAV 2014.* Springer, 646–662. https://doi.org/10.1007/978-3-319-08867-9_43

Leonid Libkin. 2003. Variable independence for first-order definable constraints. *ACM Trans. Comput. Log.* 4, 4 (2003), 431–451. http://doi.acm.org/10.1145/937555.937557

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. https://doi.org/10.1145/2837614.2837641

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

Yasuhiko Minamide. 2005. Static approximation of dynamically generated Web pages. In *Proceedings of the 14th international conference on World Wide Web, WWW 2005.* ACM, 432–441. https://doi.org/10.1145/1060745.1060809

Christophe Morvan. 2000. On Rational Graphs. In *Foundations of Software Science and Computation Structures, Third International Conference, FOSSACS 2000.* Springer, 252–266. https://doi.org/10.1007/3-540-46432-8_17

Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. 2004. Abstract DPLL and Abstract DPLL Modulo Theories. In *Logic for Programming, Artificial Intelligence, and Reasoning, 11th International Conference, LPAR 2004 (LNCS),* Vol. 3452. Springer, 36–50. https://doi.org/10.1007/978-3-540-32275-7_3

Nicholas Pippenger. 2010. *Theories of Computability.* Cambridge University Press.

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

Philipp Rümmer. 2008. A Constraint Sequent Calculus for First-Order Logic with Linear Integer Arithmetic. In *Logic for Programming, Artificial Intelligence, and Reasoning, 15th International Conference, LPAR 2008, LNCS 5330.* Springer, 274–289. https://doi.org/10.1007/978-3-540-89439-1_20

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.* IEEE, 513–528. https://doi.org/10.1109/SP.2010.38

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. ACM, 488–498. https://doi.org/10.1145/2491411.2491447

Koushik Sen, Darko Marinov, and Gul Agha. 2005. CUTE: a concolic unit testing engine for C. In *Proceedings of the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2005, Lisbon, Portugal, September 5-9, 2005, ESEC/SIGSOFT FSE 2005*. ACM, 263–272. https://doi.org/10.1145/1081706.1081750

Minh-Thai Trinh, Duc-Hiep Chu, and Joxan Jaffar. 2014. S3: A Symbolic String Solver for Vulnerability Detection in Web Applications. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, CCS 2014*. ACM, 1232–1243. https://doi.org/10.1145/2660267.2660372

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. https://doi.org/10.1007/978-3-319-41528-4_12

Andrew van der Stock, Brian Glas, Neil Smithline, and Torsten Gigler. 2017. OWASP Top 10 – 2017. https://www.owasp.org/index.php/Top_10-2017_Top_10. Referred January 2018.

Margus Veanes, Nikolaj Bjørner, Lev Nachmanson, and Sergey Bereg. 2017. Monadic Decomposition. *J. ACM* 64, 2 (2017), 14:1–14:28. https://doi.org/10.1145/3040488

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

Joel Weinberger, Prateek Saxena, Devdatta Akhawe, Matthew Finifter, Eui Chul Richard Shin, and Dawn Song. 2011. A Systematic Analysis of XSS Sanitization in Web Application Frameworks. In *Computer Security - ESORICS 2011 - 16th European Symposium on Research in Computer Security, Leuven, Belgium, September 12-14, 2011. Proceedings*. Springer, 150–171. https://doi.org/10.1007/978-3-642-23822-2_9

Fang Yu, Muath Alkhalaf, and Tevfik Bultan. 2010. Stranger: An Automata-Based String Analysis Tool for PHP. In *Tools and Algorithms for the Construction and Analysis of Systems, 16th International Conference, TACAS 2010*. Springer, 154–157. https://doi.org/10.1007/978-3-642-12002-2_13 Benchmark can be found at http://www.cs.ucsb.edu/~vlab/stranger/.

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. https://doi.org/10.1007/s10703-013-0189-1

Bohdan Zelinka. 1981. Graphs of semigroups. *Časopis pro pěstování matematiky* 106, 4 (1981), 407–408. http://eudml.org/doc/19323

Yunhui Zheng, Vijay Ganesh, Sanu Subramanian, Omer Tripp, Julian Dolby, and Xiangyu Zhang. 2015. Effective Search-Space Pruning for Solvers of String Equations, Regular Expressions and Length Constraints. In *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part I*. Springer, 235–254. https://doi.org/10.1007/978-3-319-21690-4_14

Yunhui Zheng, Xiangyu Zhang, and Vijay Ganesh. 2013. Z3-str: a Z3-based string solver for web application analysis. In *Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, ESEC/FSE 2013, Saint Petersburg, Russian Federation, August 18-26, 2013*. ACM, 114–124. https://doi.org/10.1145/2491411.2491456