The Power of Regular Constraint Propagation

MATTHEW HAGUE, Royal Holloway University of London, UK
ARTUR JEŻ, University of Wroclaw, Poland
ANTHONY WIDJAJA LIN, RPTU Kaiserslautern-Landau, Germany and MPI-SWS, Germany
OLIVER MARKGRAF, RPTU Kaiserslautern-Landau, Germany
PHILIPP RÜMMER, University of Regensburg, Germany and Uppsala University, Sweden

The past decade has witnessed substantial developments in string solving. Motivated by the complexity of string solving strategies adopted in existing string solvers, we investigate a simple and generic method for solving string constraints: regular constraint propagation. The method repeatedly computes pre- or post-images of regular languages under the string functions present in a string formula, inferring more and more knowledge about the possible values of string variables, until either a conflict is found or satisfiability of the string formula can be concluded. Such a propagation strategy is applicable to string constraints with multiple operations like concatenation, replace, and almost all flavors of string transductions. We demonstrate the generality and effectiveness of this method theoretically and experimentally. On the theoretical side, we show that RCP is sound and complete for a large fragment of string constraints, subsuming both straight-line and chain-free constraints, two of the most expressive decidable fragments for which some modern string solvers provide formal completeness guarantees. On the practical side, we implement regular constraint propagation within the open-source string solver OSTRICH. Our experimental evaluation shows that this addition significantly improves OSTRICH's performance and makes it competitive with existing solvers. In fact, it substantially outperforms other solvers on random PCP and bioinformatics benchmarks. The results also suggest that incorporating regular constraint propagation alongside other techniques could lead to substantial performance gains for existing solvers.

CCS Concepts: • **Theory of computation** → **Automated reasoning**; **Program verification**; **Program analysis**; *Logic and verification.*

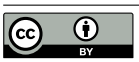
Additional Key Words and Phrases: string constraints, SMT, constraint propagation, program analysis

ACM Reference Format:
Matthew Hague, Artur Jeż, Anthony Widjaja Lin, Oliver Markgraf, and Philipp Rümmer. 2025. The Power of Regular Constraint Propagation. *Proc. ACM Program. Lang.* 9, OOPSLA2, Article 387 (October 2025), 29 pages. https://doi.org/10.1145/3763165

1 Introduction

Strings are a fundamental data type in many programming languages, heavily utilized across popular languages like Python and JavaScript. These languages are often equipped with rich string libraries, enabling a programmer to "do more with less code". However, string manipulation is error-prone, leading to security vulnerabilities such as cross-site scripting (XSS) [owa 2013, 2017,

Authors' Contact Information: Matthew Hague, Royal Holloway University of London, Department of Computer Science, Egham, UK, matthew.hague@rhul.ac.uk; Artur Jeż, University of Wroclaw, Institute of Computer Science, Wroclaw, Poland, aje@cs.uni.wroc.pl; Anthony Widjaja Lin, RPTU Kaiserslautern-Landau, Kaiserslautern, Germany and MPI-SWS, Kaiserslautern, Germany, awlin@mpi-sws.org; Oliver Markgraf, RPTU Kaiserslautern-Landau, Kaiserslautern, Germany, markgraf@cs.uni-kl.de; Philipp Rümmer, University of Regensburg, Regensburg, Germany and Uppsala University, Uppsala, Sweden, philipp.ruemmer@ur.de.



ACM 2475-1421/2025/10-ART387
https://doi.org/10.1145/3763165

2021]. Recent years have witnessed the emergence of string solving as a promising automated method for reasoning about string-manipulating programs, among others, the presence of security vulnerabilities like XSS. String solving has the theory of word equations as its foundation, a topic studied for over 50 years. While many decidability results have been established (e.g., [Abdulla et al. 2019; Chen et al. 2019; Jeż 2016; Plandowski 1999]), their underlying proofs are often intricate and remain difficult to implement in practice. The field also continues to face several long-standing open problems.

On the practical side of string solving, substantial progress has been made in recent years, with solvers adopting and optimizing foundational algorithms. A prevalent method remains the strategy of *splitting word equations* into multiple simpler equations, often combined with delayed handling of regular expressions [Makanin 1977; Nielsen 1917], and implemented in modern tools such as Z3, Z3-alpha, Z3-Noodler, and OSTRICH [Chen et al. 2019, 2024; de Moura and Bjørner 2008; Lu et al. 2024]. This method, though effective, is far from straightforward: achieving good performance requires carefully crafted heuristics that determine the most efficient way to split equations and choose the branches to explore first, balancing the complexity and efficiency of the solving process. This is usually achieved by a combination of other reasoning rules (e.g. involving length, regular languages, etc.). Equation splitting is a fundamental method used by most existing string solvers.

While equation splitting has been the dominant strategy so far, an alternative technique grounded in constraint propagation (CP) [Rossi et al. 2006] has also developed over time. The first major application of this approach in string constraint solving can be traced back approximately 20 years ago [Golden and Pang 2003], with early work on regular constraint propagation (RCP) techniques over fundamental string functions such as length, concatenation, prefix, suffix and regular constraints. RCP, works by iteratively computing pre-images or post-images of regular languages under the string functions present in a formula, propagating regular constraints forwards and backwards. Propagation deduces increasingly more precise information about the possible values of string variables, until it either encounters a conflict, indicating that the formula is unsatisfiable, or reaches a point where the satisfiability of the string formula can be determined.

Over time, various solvers have incorporated RCP techniques to varying degrees, including STRANGER [Yu et al. 2010], CertiStr [Kan et al. 2022], OSTRICH [Chen et al. 2022, 2019], and Z3-Noodler[Blahoudek et al. 2023; Chen et al. 2023a, 2024; Havlena et al. 2024]. However, these tools either lack completeness guarantees or support only restricted fragments.

Our work builds upon these prior efforts by refining and extending RCP techniques. We demonstrate that a subset of the proof system is sufficient to achieve completeness in the chain-free fragment, while also capturing more complex string functions such as `replaceAll`, transducers, reverse operations, and polyregular functions. Additionally, we provide an intuitive, algorithmic characterization of the chain-free fragment and analyze the limitations of the proof system, even when augmented with additional proof rules like equation splitting.

Contributions. We formalize RCP as a subset of the proof system from [Chen et al. 2022]. Our **first contribution** is to show that RCP is complete for a large natural fragment of string constraints, which we call the *orderable fragment*. This fragment subsumes two of the largest known decidable fragments called the *chain-free string constraint fragment* [Abdulla et al. 2019] (over concatenation and rational transducer functions) and the *straight-line fragment* [Chen et al. 2019] (over a more general class of string functions satisfying some semantic conditions). In doing so, we discover a substantially simplified formulation of the chain-free fragment in [Abdulla et al. 2019]. Our **second contribution** is the implementation of RCP within the state-of-the-art solver OSTRICH [Chen et al. 2023b]. We demonstrate a significant improvement in OSTRICH's performance, and

our experiments indicate that our RCP-based solver is the only string solver capable of solving random PCP benchmarks. Moreover, when combined with other solvers, our approach reduces the number of unsolved benchmarks by improvements ranging from 25% up to 74%, thereby complementing existing techniques. Finally, given the power of regular constraint propagation, it is natural to wonder if it can prove unsatisfiability for all unsatisfiable string constraints with only word equations and regular constraints, which would give rise to a simple alternative for complete algorithms (e.g. Makanin's algorithm [Makanin 1977] and recompression algorithm [Jeż 2016]). Our **third contribution** is a result exhibiting this theoretical limitation of RCP, even together with some other commonly used proof rules. In particular, we enrich RCP with two simple proof rules: (1) Nielsen's transformation rule (which can be used to show decidability quadratic word equations [Diekert 2002]), and (2) The "Cut" rule [Chen et al. 2022]. This essentially amounts to the proof system of [Chen et al. 2022] enriched with Nielsen's transformation. We show that there are string constraints with concatenation and regular constraints that are unsatisfiable but cannot be proved using only RCP, Nielsen, and Cut. To the best of our knowledge, this constitutes one of the handful of impossibility results for a string solving strategy, especially with a rich set of proof rules.

Organization. The paper is structured as follows. Section 2 gives three motivating examples. Section 3 formalizes regular constraint propagation as a proof system. Section 4 discusses completeness and decidability. Theoretical limitations are addressed in Section 5. Finally, Section 6 presents experimental results, Section 7 discusses related work and Section 8 concludes with remarks on future work.

2 Motivating Examples

2.1 String Sanitization

To motivate the challenges in reasoning about real-world string-processing programs, we consider a transformation that normalizes decimal strings by first applying sanitization and then rewriting and restructuring the content.

Consider the function shown in Figure 1, which takes a string representation of a decimal number, removes leading and trailing whitespace, eliminates redundant zeros, and ensures a canonical form. The function would, for instance, normalize `" 000123.45000 "` to `"123.45"`. This function can be fully expressed as a string constraint and solved within a constraint-solving framework. By adding an assertion that the output must be in normalized form, a solver can determine whether all inputs correctly transform into their canonical representation. This enables automated reasoning about the correctness and behavior of the function, allowing us to verify that any input will always yield a properly formatted decimal number.

It should be noted that the string transformations in Figure 1 are challenging for existing SMT solvers. With the most widely used SMT solvers, Z3 [de Moura and Bjørner 2008] and cvc5 [Barbosa et al. 2022], the `decimal.match` step could only be modeled using the operations `str.indexof` and `str.substr`. Z3, cvc5 and Z3-noodler are not complete for string constraints involving those operations, however, so that no guarantees can be given that the solvers are able to verify desired properties of the `normalize` function. Related work implemented in the SMT solver OSTRICH [Chen et al. 2022] supports similar functionality using prioritized streaming transducers. These transducers can express the `decimal.match` operation, but result in a more complex and comparatively slower solving procedure. We show that RCP gives rise to a much simpler decision procedure that is still able to precisely model the transformations in Figure 1.

The function can be expressed in terms of functional transformations. The first step applies a trimming function f_{trim}, which removes leading and trailing whitespace. The result is then

```
function normalize(decimal) {
  decimal = decimal.trim();
  const decimalReg = /^(\d+)\.?( \d*)$/;
  var decomp = decimal.match(decimalReg);
  var result = "";
  if (decomp) {
    var integer = decomp[1].replace(/^0+/, "");
    var fractional = decomp[2].replace(/0+$/, "");
    if (integer !== "") result = integer; else result = "0";
    if (fractional !== "") result = result + "." + fractional;
  }
  return result;
}

```

Fig. 1. Normalize a decimal by trimming whitespace and removing leading and trailing zeros.

decomposed into an integer part y and a fractional part z, separated by a decimal point. The transformation proceeds by applying two additional functions: g_{lead} to remove leading zeros from y and g_{trail} to remove trailing zeros from z.

This transformation can be expressed using the following constraints:

$$\begin{aligned}
&\text{decimal} \in \text{decimalReg} \wedge \\
&\text{decimal} = f_{\text{trim}}(\text{input}) \wedge \\
&\text{decimal} = \text{integer} \mathbin{+\!\!+} \text{"."} \mathbin{+\!\!+} \text{fractional} \ \wedge \\
&\text{result} = g_{\text{lead}}(\text{integer}) \mathbin{+\!\!+} \text{"."} \mathbin{+\!\!+} g_{\text{trail}}(\text{fractional}) \wedge \\
&\text{result} \notin \text{correctFormat}
\end{aligned}$$

where f_{trim}, g_{lead}, and g_{trail} can all be expressed as functional transducers. In this formulation, we use ++ to denote string concatenation, and assert that the result string does not match some regular expression "correctFormat" to verify that no execution exists in which string normalization fails.

The constructed formula is unsatisfiable, but beyond the fragments decided by most of today's string solvers. Solvers like Z3, cvc5, or Z3-noodler are not able to handle f_{trim}, g_{lead}, and g_{trail} and again have to resort to functions like `str.indexof` and `str.substr` to encode those transformations. The solver OSTRICH [Chen et al. 2022, 2019] supports reasoning over transducers and `replace`/`replaceAll`, but requires input formulas in the *straight-line* fragment to guarantee completeness. Our formula is not straight-line, since two equations with "decimal" as the left-hand side exist, causing OSTRICH to fail in solving the constraint. We will show that the formula is in a new fragment proposed in this paper, denoted the *orderable* fragment, which generalizes both the straight-line and chain-free fragments. Since RCP is complete for orderable formulas, it gives rise to a decision procedure for formulas like the one shown here and can easily show the formula to be unsatisfiable. We revisit this example in Section 6.2, Table 1, where RCP is the only solver to handle it successfully.

2.2 Post's Correspondence Problem

Post's Correspondence Problem (PCP) is a well-known *undecidable problem* that asks whether a given set of *domino-like word pairs* (tiles) can be arranged in sequence such that the concatenation of the top and bottom sequences produces the same string.

In general, PCP instances consist of *multiple dominos*, and the word lengths on either side of a domino may differ. However, even for highly restricted cases, the problem remains undecidable.

To illustrate, consider the following *simplified instance*, consisting of a *single* domino: $\frac{10}{01}$. This means we are given the tile $\frac{10}{01}$ and need to determine whether any non-empty sequence of these tiles produces the same string on both top and bottom.

In string constraint solving, we encode this problem using a *string variable* x that selects repeated instances of this domino. Specifically:

$$\mathrm{x} \in \texttt{"2"}^+ \wedge \mathrm{y} = \texttt{replaceAll}(\mathrm{x}, \texttt{"2"}, \texttt{"10"}) \wedge$$
$$\mathrm{z} = \texttt{replaceAll}(\mathrm{x}, \texttt{"2"}, \texttt{"01"}) \wedge \mathrm{y} = \mathrm{z}$$

Here, x is a *non-empty string* over the symbol `"2"` (denoted $\texttt{"2"}^+$), which represents repeated selections of the given domino. The `replaceAll` function *maps* each occurrence of `"2"` to its corresponding *top (*`"10"`*) and bottom (*`"01"`*)* strings. This formulation enforces that applying the same sequence of replacements to both top and bottom strings must yield the same result. However, this instance is unsatisfiable—no sequence of replacements can equate the top and bottom string. By propagating the regular constraints from x forward onto y and z, and using the fact that $y = z$, we find that any solution for y and z must lie in $(\texttt{"10"})^+ \cap (\texttt{"01"})^+$. Since this intersection is empty, the formula is unsatisfiable. Notably, despite its simplicity, this small PCP instance is already beyond the reach of state-of-the-art string solvers. See Table 1 in Section 6.2 for solver performance on this example.

2.3 Reverse Transcription in Bioinformatics

This example models a reverse transcription process inspired by bioinformatics [Compeau and Pevzner 2015; Mount 2004]. Here, an unknown RNA string y is converted into a DNA string by applying a series of `replaceAll` operations that simulate nucleotide base pairing. In addition, the RNA string is required to contain a specific pattern.

The following formula captures an instance of this problem (where the `...`, as in `"TGAGTAT..."` and `"ucuc..."`, indicate longer concrete strings omitted for brevity):

$$\mathrm{x} = \texttt{"TGAGTAT..."} \wedge$$
$$\mathrm{y1} = \texttt{replaceAll}(\mathrm{y}, \texttt{"u"}, \texttt{"A"}) \wedge$$
$$\mathrm{y2} = \texttt{replaceAll}(\mathrm{y1}, \texttt{"a"}, \texttt{"T"}) \wedge$$
$$\mathrm{y3} = \texttt{replaceAll}(\mathrm{y2}, \texttt{"g"}, \texttt{"C"}) \wedge$$
$$\mathrm{x} = \texttt{replaceAll}(\mathrm{y3}, \texttt{"c"}, \texttt{"G"}) \wedge$$
$$\mathrm{z} = \texttt{"ucuc..."} \wedge$$
$$\texttt{str.contains}(\mathrm{y}, \mathrm{z})$$

In this instance, the solver must determine an RNA string y such that, after sequentially replacing `"u"` with `"A"`, `"a"` with `"T"`, `"g"` with `"C"`, and `"c"` with `"G"`, the resulting DNA string x matches a given constant. Moreover, y must contain the RNA pattern specified by z, as enforced by `str.contains`(y, z). This example illustrates another application of complex string transformations in a biologically motivated context. Importantly, this instance falls within the straight-line fragment and can be solved by OSTRICH as well as by our solver. However, it also highlights the need for robust support for complex string functions such as `replaceAll`—a feature that receives only limited support in other state-of-the-art solvers. Solver performance on one instance of this benchmark is summarized in Table 1 in Section 6.2.

3 Regular Constraint Propagation as a Proof System

3.1 String Constraint Fragment

In this work, we focus on a subset of string constraint language that is most relevant for regular constraint propagation. In particular, we do not deal with length constraints, which can already be effectively dealt with *after* regular constraint propagation is done, for instance, by looking at the length abstraction of the equational constraints, regular constraints, together with the length constraints (e.g. see [Chen et al. 2023b,a; Lin and Barceló 2016]). For simplicity, we also assume that formulas are provided in a *normal form*. The syntax of this normal form is defined as follows:

$$\psi \ ::= \ \phi \mid \psi \wedge \psi \qquad\qquad \phi \ ::= \ x \in e \mid x = f(x_1, \ldots, x_n)$$

This normal form is not restrictive, as any formula in the supported string language can be systematically transformed into an equivalent formula in this normal form.

To clarify the components of this normal form, we denote string variables by $x, x_1, \ldots, x_n$ (and later using y and z). A string constraint ψ is a conjunction of string formulas ϕ. There are two types of string formulas: *regular constraints* and *equational constraints*.

Regular constraints are regular membership tests $x \in e$ that assert that the value assigned to x must belong to the regular language defined by a regular expression (RegEx) e. The exact supported regular expression syntax is not expanded on here, as it does not affect decidability. However, it may influence the applicability or efficiency of certain proof rules discussed later, depending on whether features like intersection or complement are supported. In an implementation, one could use equivalent representations of regular languages including (nondeterministic) finite automata.

Equational constraints are formulas $x = f(x_1, \ldots, x_n)$ that assert that the value of x is equal to the result of applying string function f to the values of $x_1, \ldots, x_n$. We admit general string functions $f : (\Sigma^*)^n \to (\Sigma^*)$ (in fact, even when f is a relation), but our approach works especially under the assumption of at least one of *forwardability* and *backwardability*, which will be expounded below. A plethora of string functions satisfy at least one of these conditions including concatenation, `replace`, `replaceAll`, `reverse`, and functions defined by various flavors of transducers (e.g. rational, two-way, and streaming transducers) (e.g. [Alur and Cerný 2010; Alur and Deshmukh 2011; Berstel 1979; Chen et al. 2018, 2019, 2023a]).

3.2 The Proof System

We introduce the *RCP proof system*, which is based on propagating regular languages. The system operates on *one-sided sequents* (i.e. left-sided) and can be interpreted similarly to *Gentzen-style sequents* [Gentzen 1935]. A *sequent* in this system is a string constraint, denoted as $\Gamma = \{\phi_1, \phi_2, \ldots, \phi_n\}$, where each ϕ_i is a string formula; it should be understood as a conjunction $\phi_1 \wedge \phi_2 \wedge \cdots \wedge \phi_n$.

For convenience, we sometimes write sequents as a list $\Gamma, \phi_1, \ldots, \phi_n$ which represents the union $\Gamma \cup \{\phi_1, \ldots, \phi_n\}$, and when writing a string constraint, we sometimes write $\{\phi_1, \phi_2\}$ instead of $\phi_1 \wedge \phi_2$. A string constraint $\phi_1 \wedge \cdots \wedge \phi_n$ naturally corresponds to a sequent $\{\phi_1, \ldots, \phi_n\}$.

A *proof* (or *proof tree*) is a *rooted tree* (V, E) where the nodes V represent sequents. For the sake of presentation, we omit an explicit labeling function from nodes to sequents and assume that each node is uniquely identified within the tree. In particular, if the same sequent appears more than once in the proof tree, we treat each occurrence as a distinct node.

The root of the tree is located at the *bottom* and corresponds to the sequent whose unsatisfiability we want to prove, while the *leaves* at the *top* of the tree are called *axioms*, and represent unsatisfiable instances. We say a proof tree is *closed* if and only if all leaves correspond to the empty sequent $\Gamma = \emptyset$. The edge relation of the proof tree is given by the proof rules. Except for the initial sequent, every sequent in the proof tree must be *derived* by one of the proof rules shown in Figure 2.

$$\texttt{[Close]}\quad \frac{}{\Gamma, x \in e_1, \ldots, x \in e_n} \text{ if } L(e_1) \cap \cdots \cap L(e_n) = \emptyset$$

$$\texttt{[Intersect]}\quad \frac{\Gamma, x \in e}{\Gamma, x \in e_1, \ldots, x \in e_n} \text{ if } L(e) = \bigcap_{i=1}^{n} L(e_i)$$

$$\texttt{[Fwd-Prop]}\quad \frac{\Gamma, x \in e, x = f(x_1, \ldots, x_n), x_1 \in e_1, \ldots, x_n \in e_n}{\Gamma, x_1 \in e_1, \ldots, x_n \in e_n, x = f(x_1, \ldots, x_n)} \text{ if } L(e) = f(L(e_1), \ldots, L(e_n))$$

$$\texttt{[Bwd-Prop]}\quad \frac{\left\{ \begin{array}{c} \Gamma, x \in e, x = f(x_1, \ldots, x_n), \\ x_1 \in e_1^i, \ldots, x_n \in e_n^i \end{array} \right\}_{i=1}^{k}}{\Gamma, x \in e, x = f(x_1, \ldots, x_n)} \begin{array}{l} \text{if } f^{-1}(L(e)) \\ = \bigcup_{i=1}^{k} (L(e_1^i) \times \cdots \times L(e_n^i)) \end{array}$$

Fig. 2. Key proof rules of RCP

A *proof rule* in our system takes the form $\frac{S_1 S_2 \ldots S_n}{S}$, indicating that the sequent S may be proved from a sequence of sequents $S_1, S_2, \ldots, S_n$. If a sequent S is derived from sequents $S_1, \ldots, S_n$ by a proof rule, then S is the parent node of $S_1, \ldots, S_n$ in the proof tree. Formally, the edge relation $E \subseteq V \times V$ is defined so that for every proof step, we have $(S, S_1), \ldots, (S, S_n) \in E$.

Our *RCP proof system* consists of exactly the proof rules, as given in Figure 2; note that $\times$ in `[Bwd-Prop]` represents the Cartesian product. Before explaining the proof rules, we note that the proof system is a fragment of the proof system introduced in [Chen et al. 2022], from which we obtain that it is *sound*.

Lemma 3.1 (Soundness [Chen et al. 2022]). *The RCP proof system is sound. That is, the root of a closed proof is an unsatisfiable sequent.*

We explain the proof rules. The `Close` rule terminates a branch when a contradiction is found. That is, a variable is subject to an unsatisfiable combination of regular constraints. The `Intersect` rule computes the intersection of regular expressions. Forwards and backwards constraint propagation derive new regular constraints on string variables that arise due to function application. Forwards propagation transmits regular constraints from function inputs to function outputs, while backwards propagation propagates constraints from function outputs back to their inputs. Details and examples are given below.

For example, when $x = \texttt{concat}(y, z)$, we can combine any known regular constraints $y \in e_1$ and $z \in e_2$ to derive that $x \in e_1 e_2$. For readability, we will from now on write $x = yz$ instead of $x = \texttt{concat}(y, z)$ to denote string concatenation. For this forward propagation, we require that the image of a function on regular languages is also regular, which is the case when concatenating two different variables. In general, the forward-propagated constraint is only an *overapproximation* of the true constraint on x: Consider $x = f(y) \wedge y \in e$, where $f(y) = yy$, in this case, the obtained regular constraint $x \in ee$ is a superset of the true constraint, which is $\{ww : w \in L(e)\}$. Although overapproximations can be easily added in our RCP implementation (and are in fact sound when proving unsatisfiability), we do not admit this in our theoretical study and hence not in our proof system above.

In the backwards direction, when $x = yz$, a regular constraint $x \in e$ can be split into a finite disjunction of conjuncts of the form $y \in e_1^i$ and $y \in e_2^i$ for any $\{e_1^i\}_{i=1}^k$ and $\{e_2^i\}_{i=1}^k$ such that $L(e) = \bigcup_{i=1}^k L(e_1^i e_2^i)$. For example, if $x \in a^*b^*$, one potential disjunction could be

$$(y \in a^* \wedge z \in a^*b^*) \vee (y \in a^*b^* \wedge z \in b^*).$$

The same in fact is true for $x = f(y) \wedge y \in e$, with $f(y) = yy$. The condition for a finite number of alternatives for e_1 and e_2 is expressed by the side-condition $f^{-1}(L(e)) = \bigcup_{i=1}^k (L(e_1^i) \times \cdots \times L(e_n^i))$ and causes branching in the proof trees (all k alternatives are considered).

As mentioned, propagations can be performed for functions that allow these in at least one of the two directions. We formalize this in terms of forwardability and backwardability. We lift string functions $f : (\Sigma^*)^k \to \Sigma^*$ to act on languages in the standard way.

Definition 3.2 (Forwardable). A function $f : (\Sigma^*)^k \to \Sigma^*$ is *forwardable* if, for all regular languages $L_1, \ldots, L_k \subseteq \Sigma^*$, the image $f(L_1, \ldots, L_k)$ is also a regular language, and there exists an algorithm that, given representations of $L_1, \ldots, L_k$, computes a representation of $f(L_1, \ldots, L_k)$.

Definition 3.3 (Backwardable). A function $f : (\Sigma^*)^k \to \Sigma^*$ is *backwardable* if, for all regular languages $L \subseteq \Sigma^*$, the preimage $f^{-1}(L)$ is a recognizable relation [Berstel 1979], and there exists an algorithm that, given a representation of L, computes a representation of $f^{-1}(L)$.

The recognizable relation in the definition corresponds exactly to the notion in `Bwd-Prop`: $R \subseteq (\Sigma^*)^n$ is recognizable, if it can be represented as $R = \bigcup_{i=1}^k L_{i,1} \times \cdots \times L_{i,k}$, with each $L_{i,j}$ being a regular language. Note that such a representation is not unique.

The following lemmas summarize conditions on concatenation functions to be forwardable.

Lemma 3.4. *Suppose $t = f(x_1, \ldots, x_n)$ is a term over the function symbol* `concat` *(possibly with string constants). Then, if each x_i appears at most once in t, then the string function associated with t is forwardable.*

The proof is a simple application of the closure of regular languages under concatenation. In the case when a variable appears multiple times in t, we have also remarked above that the associated function is generally not forwardable. That said, string functions using unrestricted concatenation and string constants are always backwardable.

Lemma 3.5. *Suppose $t = f(x_1, \ldots, x_n)$ is a term over the function symbol* `concat` *(possibly with string constants). Then, the string function associated with t is backwardable.*

Proof of this lemma is folklore (and already exploited, e.g., in [Chen et al. 2019, 2023a]). Other functions are also known to be forwardable including the string-reverse function, `replaceAll(x,v,w)` — which replaces every occurrence of the constant string $v \in \Sigma^*$ in x by the constant string $w \in \Sigma^*$ — and more generally *rational transducers* (e.g. see [Berstel 1979]), which are standard finite automata with input and output (i.e. each transition has a label $(v, w) \in \Sigma^* \times \Sigma^*$, and has the semantics of consuming v in the input and then outputting w). The same is no longer true for `replaceAll(x,v,y)` (i.e. where the replacement string is a variable) [Chen et al. 2018], and two-way transducers (i.e. head of the input/output tapes can move left/right), or equivalently, streaming transducers [Alur and Cerný 2010; Alur and Deshmukh 2011]. Interestingly, all of these aforementioned functions are backwardable [Chen et al. 2019].

Example 3.6. Given the string constraint:

$$y = zu \wedge y = xx \wedge z \in b \wedge u \in a\,.$$

we apply the proof rule `Fwd-Prop` to propagate the regular constraints $z \in b$ and $u \in a$ through the equational constraint $y = zu$, allowing us to deduce that $y \in ab$.

$$\dfrac{\dfrac{\dfrac{}{\Gamma, y = xx, y \in ba, \mathbf{x} \in \mathbf{b}, \mathbf{x} \in \mathbf{a}}\,\texttt{[Close]} \quad \dfrac{}{\Gamma, y = xx, y \in ba, \mathbf{x} \in \epsilon, \mathbf{x} \in \mathbf{ba}}\,\texttt{[Close]}}{y = zu, y = xx, z \in b, u \in a, \mathbf{y} \in \mathbf{ba}}\,\texttt{[Bwd-prop]}\ y = xx}{y = zu, y = xx, z \in b, u \in a}\,\texttt{[Fwd-prop]}\ y = zu$$

Fig. 3. Proof tree for Example 3.6

Next, applying `Bwd-Prop` on $y \in ab$, we propagate this information backwards through $y = xx$. This leads to a contradiction in all branches of the proof. The proof tree illustrating this contradiction is shown in Figure 3.

4 Completeness and Decidability

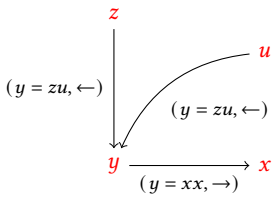


Fig. 4. Flow graph of $Eqs = \{y = zu, y = xx\}$ depicting flow of constraints.

Here we provide a general completeness condition for RCP for proving unsatisfiability. By *complete*, we mean that, for every unsatisfiable string constraint, there exists a closed RCP proof. In general, this suffices to guarantee decidability since if the constraint is satisfiable, then—by a standard argument in computability theory, based on the finiteness of the alphabet and the number of variables—one can enumerate all assignments of increasing length until a satisfying model is eventually found; otherwise, a closed RCP proof of unsatisfiability will be discovered. Our completeness condition can be understood in terms of finding a "flow" of regular constraints through forwardable and backwardable functions. In a set $Eqs = \{x_1 = f_1(y_1 \dots y_k), \dots\}$ of equational constraints, a variable x that occurs *only once* in Eqs (say in equation $\phi \in Eqs$, on either left- or right-hand side) is a potential "source point" of a flow. That is, the *only* way x can affect other variables is via the equation ϕ. Therefore, by propagating *all* the regular constraints on x through ϕ to other variables in ϕ, we may *eliminate* x, as well as ϕ, from Eqs, while preserving satisfiability. As an example, consider $Eqs = \{y = zu, y = xx\}$ along with regular constraints $z \in a^+ \wedge u \in b^+$. Since z and u occur only once and $f(z, u) := zu$ is forwardable, we could order the flow of the regular constraints $z \in a^+ \wedge u \in b^+$ in the "forward" direction of $y = f(z, u)$ resulting in the equisatisfiable constraint:

$$y = xx \wedge y \in a^+b^+.$$

In turn, since $g(x) := xx$ is a backwardable function and y occurs only once, we could order the flow of the regular constraint $y \in a^+b^+$ in the "backward" direction resulting in the equisatisfiable constraint $\bot$. That is, we found the *flow sequence* $(y = zu, \leftarrow), (y = xx, \rightarrow)$, where each pair depicts the equational constraint and the direction of propagation. The propagation steps in a flow sequence can be visualized using a *flow graph*. Given a flow sequence, we construct a directed graph where nodes represent variables. Each propagation rule (ϕ, d), with ϕ an equation like $y = f(x_1, \dots, x_k)$, contributes edges from the variables on the source side of the propagation to those on the target

side—i.e., from $x_1, \ldots, x_k$ to y for a forward step, or from y to $x_1, \ldots, x_k$ for a backward step. This flow graph helps visualize how information is propagated through equations. Notably, regular constraints do not influence the graph's structure, as it is determined solely by the equational relationships. An example of the flow graph for the constraint above is shown in Figure 4. This intuition is formalized in the Marking Algorithm (Algorithm 1) and the condition of *orderable* set of equations, as given below.

Algorithm 1: Marking Algorithm

Data: A set of equational constraints $Eqs = \{x_1 = f_1(y_1 \ldots y_k), \ldots\}$
Result: A flow sequence on the equational constraints, or $\perp$ if the constraints cannot be ordered

```
remainingEqs ← Eqs;
markedVars ← ∅;
flowSeq ← ϵ;
repeat
    foreach variable x appearing exactly once in remainingEqs do
        markedVars ← markedVars ∪ {x}
    foreach φ = (x = f(y_1, ..., y_k)) ∈ remainingEqs do
        if x ∈ markedVars and f is backwardable then
            flowSeq ← flowSeq, (φ, →);
            remainingEqs ← remainingEqs \ {φ};
        else if y_1, ..., y_k ∈ markedVars and f is forwardable then
            flowSeq ← flowSeq, (φ, ←);
            remainingEqs ← remainingEqs \ {φ};
until markedVars is unchanged;
return flowSeq if remainingEqs = ∅ else ⊥
```

4.1 Orderable Constraints

Given a set of equational constraints *Eqs*, a *flow sequence* is a sequence of pairs (ϕ, d) where $\phi \in Eqs$ and $d \in \{\leftarrow, \rightarrow\}$. Each such pair is called a *propagation rule*. A pair $(\phi, \leftarrow)$ indicates that the rule `Fwd-Prop` should be applied against the equational constraint ϕ. Conversely, $(\phi, \rightarrow)$ indicates a use of the `Bwd-Prop` rule.

Definition 4.1. A set *Eqs* of equations is said to be *orderable* if, on input *Eqs*, Algorithm 1 outputs a flow sequence. A string constraint φ is orderable if the equational part is orderable.

It can be easily checked that $\{y = zu, y = xx\}$ is orderable. An example of a set of equations that is not orderable is $\{y = zu, y = xx, y = uv\}$; the reason being that u occurs on the right hand side of two equations,

The main result of this section is that the flow sequence returned by the Marking Algorithm gives a strategy to apply the forward and backward propagation in RCP that is sound and complete.

Theorem 4.2. *Given a set of orderable constraints applying the forward propagation and backward propagation by RCP according to the flow sequence returned by Algorithm 1 is sound and complete. In particular, if such a string constraint is not satisfiable, the proof system will produce a proof with all closed branches.*

In particular, if the instance of orderable constraints is not satisfiable, RCP can produce a proof of that.

Corollary 4.3. *Every unsatisfiable orderable constraint admits a proof in RCP system with the following rules:* `Close`, `Intersect`, `Fwd-Prop`, *and* `Bwd-Prop`.

To prove Theorem 4.2 we use the following Lemmas 4.4 and 4.5, which show that the removal of the equational constraint by Algorithm 1 yields an equisatisfiable string constraint, in cases of forward and backwards propagation, respectively. Note that the proof system does not actually remove the equations.

Lemma 4.4. *If $z = f(y_1, \ldots, y_n)$ is an equational constraint in a string constraint and $y_1, \ldots, y_n$ have exactly one occurrence in the equational part of string constraint and f is forwardable then the string constraint obtained by propagating the constraints on $y_1, \ldots, y_n$ forward and removal of the equation are equisatisfiable. Formally*

$$\exists z \exists y_1, \ldots, y_n \ \phi(\bar{x}, z) \wedge z \in L_z \wedge z = f(y_1, \ldots, y_n) \wedge \bigwedge_{i=1}^{n} y_i \in L_i$$
$$\equiv \exists z \exists y_1, \ldots, y_n \ \phi(\bar{x}, z) \wedge z \in L_z \wedge z \in f(L_1, \ldots L_n) \wedge \bigwedge_{i=1}^{n} y_i \in L_i$$

Lemma 4.5. *If $z = f(y_1, \ldots, y_n)$ is an equational constraint in a string constraint and z has no other occurrence in equational part of the string constraint, then the string constraint obtained by propagating the regular constraints on z backwards and removal of the equation are equisatisfiable. Formally:*

$$\exists z \exists y_1, \ldots, y_n \ \phi(\bar{x}, y_1, \ldots, y_n) \wedge z \in L_z \wedge z = f(y_1, \ldots, y_n)$$
$$\equiv \bigvee_{j=1}^{k} \exists z \exists y_1, \ldots, y_n \ \phi(\bar{x}, y_1, \ldots, y_n) \wedge z \in L_z \wedge \bigwedge_{i=1}^{n} y_i \in L_{i,j}$$

where $f^{-1}(L_z) = \bigcup_{j=1}^{k} L_{1,j} \times \cdots \times L_{n,j}$.

Proof of Theorem 4.2. By Lemma 3.1 RCP is sound in general, so in particular it is sound for some specific order of applying the proof rules.

By assumption that the constraints are orderable we obtain that Algorithm 1 terminates with a flow sequence that gives an order on equational constraints (and information, whether we should propagate forwards or backwards). We will propagate them in this order.

Note, that applying the backwards propagation creates many branches in the proof tree and when constructing the proof tree we need to take care of each open branch. However, we treat all open branches in the same way: we will apply the same rule to each open branch (and implicitly close branches on which contradiction was obtained). We apply the intersection rule for each variable, before each of the propagations (on each open branch), without mentioning it explicitly.

Observe that the proof rules do not modify nor remove the equational constraints. Therefore, the set of equational constraints is the same in each sequent, except for the axioms. However, as a tool of the construction, we mark some equational constraints in the open branches, and as an invariant:

- in each of the open branches the same set of equations is marked and it is the same as the set of equations remainingEqs from Algorithm 1 at the corresponding step

- after application of a rule, which changes the sequent Γ to $\{\Gamma_i\}_{i=1}^k$, the constraint obtained by removing the unmarked constraints from Γ is equivalent to the (disjunction of) constraints obtained by removing the unmarked constraints from each Γ_i.

In particular, a variable occurs once in a marked equational constraints in some open sequent if and only if it occurs once in a marked equational constraints in each of the open sequents.

The proof follows by applying the propagation according to the order returned by Algorithm 1. Note that the above invariants are clearly satisfied at the beginning, when the root is the open sequent and we set all equations as marked and the Algorithm 1 has not removed any of the equational constraints.

First, consider `Intersect` rule, which rewrites sequent Γ as Γ'. We mark exactly the same set of equational constraints in Γ' as they are marked in Γ. Also none equational constraints were changed. Hence the first invariant clearly holds. For the second invariant note that we replaced some regular constraints $x \in e_1, \ldots, x \in e_k$ with $x \in e$, where $L(e) = \bigcap_{i=1}^k L(e_i)$, so clearly those are equivalent and so the second invariant holds as well.

Consider the next equational constraint for the propagation (in the sequence returned by Algorithm 1). If we are to propagate forwards for $z = f(y_1, \ldots, y_n)$, then variables $y_1, \ldots, y_n$ appear only once in the set of equational constraints in remainingEqs and so in each of the open sequents they appear exactly once in the marked equations (by the invariant). Hence the string constraint (in an open sequent and restricted to marked equational constraints) is of the form as in Lemma 4.4, so of a form

$$\Gamma, \Gamma', y_1 \in e_1, \ldots, y_n \in e_n, z = f(y_1, \ldots, y_n) \ ,$$

where Γ' are the unmarked equational constraints. Let the corresponding formula be

$$\exists z \exists y_1, \ldots, y_n \ \phi(\bar{x}, z) \wedge z \in L_z \wedge z = f(y_1, \ldots, y_n) \wedge \bigwedge_{i=1}^{n} y_i \in L_i \ ,$$

where $\phi(\bar{x}, z)$ is the concatenation of all constraints in Γ. Using the Lemma 4.4 we can remove the equational constraint obtaining an equisatisfiable formula

$$\exists z \exists y_1, \ldots, y_n \ \phi(\bar{x}, z) \wedge z \in L_z \wedge z \in f(L_1, \ldots L_n) \wedge \bigwedge_{i=1}^{n} y_i \in L_i \ . \tag{1}$$

On the other hand, the proof system applies the `Fwd-Prop` rule, i.e.

$$\frac{\Gamma, \Gamma', z \in e, z = f(y_1, \ldots, y_n), y_1 \in e_1, \ldots, y_n \in e_n}{\Gamma, \Gamma', y_1 \in e_1, \ldots, y_n \in e_n, z = f(y_1, \ldots, y_n)} \ ,$$

where e denotes a regular expression such that $L(e) = f(L(e_1), \ldots, L(e_n))$. In the new sequent we additionally unmark $z = f(y_1 \ldots, y_n)$. Clearly, in all open branches the same sets of equational constraints are marked (as we unmark exactly the same equation $z = f(y_1 \ldots, y_n)$) and this the set that remain in Algorithm 1.

It is left to observe that the sequent without the unmarked constraints $\Gamma, z \in e, y_1 \in e_1, \ldots, y_n \in e_n$ corresponds exactly to formula from (1), which we already know is equivalent to formula corresponding to old sequent without marked constraints. Hence, the second invariant was shown, which ends the proof for `Fwd-Prop`.

So consider the case when the equational constraint $z = f(y_1, \ldots, y_n)$ is scheduled for backwards propagation. Then z appears only once in the set of equational constraints in remainingEqs and so in each of the open sequents it appears exactly once in the marked equations (by the invariant). Fix an open sequent $\Gamma, \Gamma', z \in e, z = f(y_1, \ldots, y_n)$, where Γ' contains all unmarked equation constraints.

Let the formula corresponding to $\Gamma, z \in e, z = f(y_1, \ldots, y_n)$ be

$$\exists z \exists y_1, \ldots, y_n \; \phi(\bar{x}, y_1, \ldots, y_n) \wedge z \in L_z \wedge z = f(y_1, \ldots, y_n)$$

where $L_z = L(e)$. Then it is of the form as in Lemma 4.5 and we can remove the equational constraint obtaining an equisatisfiable formula:

$$\bigvee_{j=1}^{k} \exists z \exists y_1, \ldots, y_n \; \phi(\bar{x}, y_1, \ldots, y_n) \wedge z \in L_z \wedge \bigwedge_{i=1}^{n} y_i \in L_{i,j} \tag{2}$$

On the other hand, the proof system applies the `Bwd-Prop` rule, i.e.

$$\frac{\left\{ \begin{array}{c} \Gamma, \Gamma', z \in e, z = f(y_1, \ldots, y_n), \\ y_1 \in e_1^i, \ldots, y_n \in e_n^i \end{array} \right\}_{i=1}^{k}}{\Gamma, \Gamma', z \in e, z = f(y_1, \ldots, y_n)}$$

where each $e_{i,j}$ is a regular expression such that $L_{i,j} = L(e_{i,j})$. In the new sequents we additionally unmark $z = f(y_1 \ldots, y_n)$. Clearly, in all open branches the same sets of equational constraints are marked (as we unmarked the same equation $z = f(y_1 \ldots, y_n)$) and this the set remainingEqs in Algorithm 1.

It is left to observe that the formula from (2) corresponds to the (disjunction of) new sequents without unmarked equational constraints $\left\{\Gamma, z \in e, y_1 \in e_1^i, \ldots, y_n \in e_n^i\right\}_{i=1}^{k}$. Which ends the proof for all rules.

As Algorithm 1 eventually removes all equations, at this step all open branches in the proof have all equational constraints unmarked, by the invariant. At each open branch we apply the `Intersect` rule for each variable. If the intersection is empty for some variable then we close the branch. If at some branch all intersections are non-empty, then we claim that initial formula is satisfiable: at this sequent we can find a substitution satisfying the regular constraints and there are no marked equational constraints in this sequent. Using the second invariant we conclude that on each sequent on the path to the root the constraint obtained by removing the unmarked constraints are satisfiable: the only needed observation is that if some sequent Γ_j is satisfiable, then also $\{\Gamma_i\}_{i=1}^{k}$ is satisfiable. In particular, in the root sequent there are only marked equations, so the whole input constraint is satisfiable, contradiction with the assumption that it is unsatisfiable. Thus, if the initial sequent is not satisfiable then we obtain that all branches are eventually closed when we follow an order of propagations returned by Algorithm 1, which shows that the proof rules are complete for unsatisfiable formulas. Note that this does not immediately yield a solution of the original string constraint for satisfiable constraints, as only equisatisfiability is guaranteed. □

4.2 Straight-Line Constraints are Orderable

The straight-line fragment of string constraints in [Chen et al. 2019] corresponds to constraints in which the equational part can be ordered with a flow sequence of the form $(\phi_1, \rightarrow), \ldots, (\phi_n, \rightarrow)$. The fragment permits all string functions that are backwardable, which is called in [Chen et al. 2019] as a RegInvRel condition. This includes a rich class of functions such as the transductions and `replaceAll` functions described in Section 3.2.[1]

Corollary 4.6. *RCP is sound and complete on straight-line constraints. In particular, every unsatisfiable straight-line constraint admits a proof in the RCP system with the following rules:* `Close`, `Intersect`, *and* `Bwd-Prop`.

[1]The straight-line fragment also permits string relations that can be effectively expressed as recognizable relations, which can be reduced to regular constraints.

4.3 Chain-Free Constraints are Orderable

We show that orderable constraints subsume the class of chain-free constraints [Abdulla et al. 2019]. In fact, when we restrict to functions allowed in the chain-free fragment then those two classes coincide; in particular: unsatisfiable chain-free constraints admit a proof in RCP.

Theorem 4.7. *A string constraint given by string terms and rational transducers, is chain-free if and only if it is orderable.*

Corollary 4.8. *RCP is sound and complete on chain-free constraints. In particular, every unsatisfiable chain-free constraint admits a proof in the RCP system with the following rules:* `Close`, `Intersect`, `Fwd-Prop`, *and* `Bwd-Prop`.

Below we define chain-free constraints and show Lemmata needed for the proof of Theorem 4.7.

Chain-free fragment. The chain-free fragment of string constraints [Abdulla et al. 2019] is defined via a splitting graph: Consider a string constraint $\psi = \bigwedge_{i=1}^{n} \phi_i$, where each ϕ_j is of a form $t_{2j-1} = \mathcal{T}(t_{2j})$, where $\mathcal{T}$ is a rational function (so given by a rational transducer) and t_{2j-1}, t_{2j} are string terms; in particular this includes an equation $t_{2j-1} = t_{2j}$. Each term t_i is a concatenation of variables $x_{i,1}, \ldots, x_{i,n_j}$ and constants, one variable can occur many times in the concatenation. We construct the splitting graph as follows:

Nodes The graph contains nodes $\{(j,i) | 1 \leq j \leq 2n, 1 \leq i \leq n_j\}$. The node $(2j-1, i)$ represents the i-th term on the left-hand side, while node $(2j, i)$ represent the i-th term on the right-hand side and they are labelled with the corresponding variables.

Edges There is an edge from a node p to node q if there exists a node p' (different from q) such that

- p and p' represents the nodes on opposite sides of the same constraint (say ϕ_i), and
- p' and q have the same label.

A chain in the graph is a sequence of edges of the form $(p_0, p_1), (p_1, p_2), \ldots, (p_n, p_0)$. A splitting graph is chain-free if it has no chains; a set of equational string constraints is chain-free if its splitting graph is chain-free; a string constraint is chain-free if its equational part is chain-free.

Note that the original definition of chain-free fragment allowed rational relations, so $t_{2j-1} \in \mathcal{T}(t_{2j})$ (or $\mathcal{T}(t_{2j-1}, t_{2j})$, depending on the preferred syntax); our approach and methods generalize to this case, yet all relevant applications and examples in the SMT-LIB benchmarks use rational functions only, so for simplicity of presentation we use only rational functions. On the other hand, the original algorithm showing the decidability of chain-free constraints [Abdulla et al. 2019] assumed that the rational transducer is length-preserving, which is not needed in the case of orderable constraints (we omit this assumption for ease of presentation).

Lemma 4.9. *A chain-free constraint ψ is orderable.*

Proof. As regular constraints do not affect whether the set of constraints is chain free, nor whether it is orderable, we ignore them for the purpose of the proof. Without loss of generality we can assume that the string constraint is in normal form, see the full version of the paper [Hague et al. 2025, Lemma A.1], let it be $\psi := \bigwedge_{j=1}^{m} \phi_j$ and $\phi_j := x_j = f(y_1, \ldots, y_{k_j})$ or $x_j = \mathcal{T}(y)$, where f is an (arbitrary) concatenation of its variables and $\mathcal{T}(y)$ is a rational transducer. Note that in general f does not need to be forwardable. Assume that the string constraint ψ is chain-free, thus the splitting graph defined by ψ has no cycles. For the sake of contradiction, assume that Algorithm 1 returns $\bot$. That is, all equational constraints in *Eqs* are eliminated. Observe that merging all nodes $(2j, i)$ over various i, so all nodes corresponding to a single r.h.s., does not create a cycle: all nodes $(2j, i)$ over various i have the outgoing edges to exactly the same nodes, so if there is an incoming edge to $(2j, i)$ and outgoing from $(2j, i')$ to p then there is also outgoing from $(2j, i)$ to p.

So let us merge all nodes corresponding to one r.h.s. to one node. Consider an equational constraint, say ϕ_j. Then the variable on its left-hand side, say x, is not marked, as otherwise f (or $\mathcal{T}$) is backwardable and so the equational constraint would have been removed. Hence this variable appears in some other equational constraint, say $\phi_{j'}$. Then the node corresponding to the right-hand side of ϕ_j has an outgoing edge (to the node representing x in $\phi_{j'}$). Moreover, the node representing the other side of $\phi_{j'}$ has an edge to the l.h.s. of ϕ_j. Observe that at least one variable of the r.h.s. of ϕ_j is not marked: if all of them were marked and the equational constraint is $x = \mathcal{T}(y)$ then $\mathcal{T}$ is forwardable and so this equational constraints should be removed, if the equational constraint is $x = f(y_1, \ldots, y_k)$ then as all y_i are marked, each occurs once in the concatenation defined by f, and so f is forwardable, so it also should be removed. Thus, one of the variables on the r.h.s. of ϕ_j appears somewhere else; using the same argument we show that the l.h.s. of ϕ_j also has at least one outgoing edge and the r.h.s. has at least one incoming edge. Thus, each node has an outgoing edge and an incoming edge, therefore the simplified splitting graph has a cycle, and so also the splitting graph has a cycle.

Note that the observation that if all variables of f are marked then f is forwardable is crucial for the argument. □

Lemma 4.10. *If a string constraint given by string terms and rational transducers is orderable then it is chain-free.*

Proof. As in Lemma 4.9 we can ignore the regular constraints.

Note that if remainingEqs is not chain-free then remainingEqs $\cup \{\phi\}$ is not chain-free: each edge in splitting graph for remainingEqs exists in the splitting graph for remainingEqs $\cup \{\phi\}$.

We show that whenever the Marking Algorithm removes an equational constraint ϕ from remainingEqs $\cup \{\phi\}$, then in the splitting graph this corresponds to a removal of some nodes without incoming edges and some nodes without outgoing edges together with the edges leading from and to them, respectively, which clearly cannot remove nor create a cycle.

So suppose that ϕ is an equational constraint $x = f(y_1, \ldots, y_k)$ or $x = \mathcal{T}(y_1)$ is removed because x is marked. Then the node corresponding to x has no incoming edges (as there is no other copy of x) and each y_i has no outgoing edges (as there is no other occurrence of x).

Finally, observe that the edges that are missing in the splitting graph for remainingEqs with respect to splitting graph remainingEqs $\cup \{\phi\}$ are the edges to nodes corresponding to $y_1, \ldots, y_k$ and edges from x.

The analysis in the case when $y_1, \ldots, y_k$ are marked are done in a similar manner (this time there are no incoming edges to each of $y_1, \ldots, y_k$ and no outgoing edge from x). □

Lemmata 4.9, Lemma 4.10 prove Theorem 4.7; Corollary 4.8 follows from Corollary 4.3.

5 Lower Bounds

We now show that RCP paired with Nielsen's transform and the Cut rule (for regular constraints, described in detail below) cannot solve some string constraints; those string constraints of course are not chain-free. The concrete example in question is

$$xyxy = yxyx \ \wedge \ x \in a^+ \ \wedge \ y \in a^*ba^* \tag{3}$$

It is easy to see that the constraints in (3) are not satisfiable, by looking at the first position of b in any candidate solution. Note that length-analysis can reduce this equation into two copies of $xy = yx$, for which Nielsen's transform can detect unsatisfiability, even without the Cut rule or regular constraint propagation.

$$\frac{\Gamma, x \in e \quad \Gamma, x \in e'}{\Gamma} \text{ if } L(e') = \Sigma^* \setminus L(e) \qquad \texttt{[Cut]}$$

$$\frac{\begin{array}{c}\Gamma[x/y], u[x/y] = v[x/y], \{y \in e\}_{e \in R_x \cup R_y} \\ \{\Gamma[x/yx], x(u[x/yx]) = v[x/yx], x \in e_x^i, y \in e_y^i\}_{i=1}^{k} \\ \{\Gamma[y/xy], u[y/xy] = y(v[y/xy]), x \in e_x^i, y \in e_y^i\}_{i=k+1}^{\ell}\end{array}}{\Gamma, \{x \in e\}_{e \in R_x}, \{y \in e\}_{e \in R_y}, xu = yv} \qquad \texttt{[Nielsen]}$$

Fig. 5. Additional proof rules.

For the sake of completeness, we give proof system rules for the Cut rule and Nielsen transformation, see Fig. 5: For the Cut rule we create two branches by adding an arbitrary regular constraint and its complement on a variable.

Nielsen's transform considers leading symbols of an equation and branches into three cases, depending on which symbol is a prefix of the other (there is also a symmetric version for the suffix). In Figure 5 the cases from top-to-bottom are: x equals y, y is a prefix of x, and finally x is a prefix of y. In particular, it makes substitutions such as $x \leftarrow yx'$ (when y is a prefix of x) and needs to compute the regular constraints for x'. After the substitution and computation of the constraints, x' is renamed to x. We use the notation $u[x/y]$ to denote that y is substituted in place of x in the term u.

The rule is somewhat similar to the rule for backward propagation, in particular it has several branches. This is not surprising, as for a fixed y we can treat yx' as function that prepends y to its argument x'. We have a regular constraint (e_x) on its output and want to compute (propagate) the constraint $e_{x'}$ on its input. In particular, we will replace the constraints e_x, e_y with a family $\{(e_y^i, e_x^i)\}$, one pair for each branch. The actual formula for $\{(e_y^i, e_x^i)\}$ can be given.[2] However, for the purpose of the lower bound we do not need this exact formula and we can in fact allow the proof system to overapproximate it — it is enough to assume "soundness of the substitution". Let e_x, e_y be the intersection of all constraints on x, y, i.e. $L(e_x) = \bigcap_{e \in R_x} L(e), L(e_y) = \bigcap_{e \in R_y} L(e)$, we assume

$$\exists x'\ x = yx' \wedge y \in L(e_y) \wedge x \in L(e_x) \implies \exists i\ y \in L(e_y^i) \wedge x' \in L(e_x^i) \tag{4}$$

i.e. we allow that the new constraints e_y^i, e_x^i overapproximate the old one. However, we require a simple upper bound where e_y^i only accepts values that also appear in e_y and e_x^i only accepts values that are correct for some valid value of y (not necessarily accepted by e_y^i). I.e. we assume

$$L(e_y^i) \subseteq L(e_y) \tag{5}$$

$$L(e_x^i) \subseteq \{v \ :\ \exists u\, uv \in L(e_x) \wedge u \in L(e_y)\} \tag{6}$$

Note that in general Nielsen's rule (as given above) does not detect a solution in which initially some variable has an empty substitution. However, such substitutions are forbidden by the regular constraints in (3).

[2]We consider DFA A recognizing $L(e_x)$ and consider the possible state q reached by A after reading y and require that A starting in q and reading x' ends in a final state; alternatively, this can be handled using matrix transition functions or the semigroup-approach to regular languages.

We say that a set A is co-finite in B, when $B \setminus A$ is finite. Consider a condition:

(1) if the regular constraint on x defines a language L_x and for y a language L_y then there is m such that L_x is co-finite in $(a^m)^*$ and there are languages $L_{y,\ell}, L_{y,r}$ co-finite in $(a^m)^*$ satisfying $L_{y,\ell} b L_{y,r} \subseteq L_y$.

Theorem 5.1. *Consider a proof system that uses: Nielsen's rule, regular constraint propagation and the Cut rule only and a proof for (3) using this system. If Condition 1 holds at some sequent, then after application of a proof rule it holds in at least one of the obtained sequents (perhaps for some other value of m).*

In this way we obtain the main result of this section:

Corollary 5.2. *A proof system that uses: Nielsen's rule, regular constraint propagation and the Cut rule only, cannot prove the unsatisfiability of (3).*

Proof. Condition (1) implies in particular that there are (infinitely many) words satisfying the regular constraints, so in particular we cannot close a sequent satisfying the condition.

Clearly, condition (1) holds for the initial regular constraints and so by Theorem 5.1 there exists an infinite path of sequents, in particular, we cannot close the whole proof, which shows the claim. □

To prove Theorem 5.1 we use two auxiliary Lemmata:

Lemma 5.3. *Given regular sets $L_x \subseteq a^*, L_y \subseteq a^*ba^*$ there is a number m such that*

$$L_x = \bigcup_i L_i \qquad \text{where } L_i \in \left\{a^{k_i}, a^{k_i}(a^m)^*\right\} \text{ for each } i$$

$$L_y = \bigcup_i L_{i,\ell} b L_{i,r} \qquad \text{where } L_{i,\ell} \in \left\{a^{k_{i,\ell}}, a^{k_{i,\ell}}(a^m)^*\right\}, L_{i,r} \in \left\{a^{k_{i,r}}, a^{k_{i,r}}(a^m)^*\right\} \text{ for each } i$$

for appropriate values $k_i, k_{i,\ell}, k_{i,r}$ (over all i). In particular, each $L_i, L_{i,\ell}, L_{i,r}$ is either finite or co-finite in $(a^m)^*$. Furthermore, if such a representation exists for m, then it exists for each multiple of m. In particular, if we are given several such sets, then there is a representation of all of them for a common m.

In case of subsets of a^* the proof follows from application of well-known characterizations of unary regular languages, say Chrobak normal form. The proof for subsets of a^*ba^* requires some additional simple arguments.

Lemma 5.4. *If regular sets $L_x \subseteq a^*, L_y \subseteq a^*ba^*$ are represented as in Lemma 5.3 for a common m and there are infinite sequences of words $x_i = a^{k_i m} \in L_x$, $y_i = a^{k_{i,\ell} m} b a^{k_{i,r} m} \in L_y$ where all k_i are pairwise different, all $k_{i,\ell}$ are pairwise different and all $k_{i,r}$ are pairwise different, then L_x is co-finite in $(a^m)^*$ and there are $L_{y,\ell}, L_{y,r}$ co-finite in $(a^m)^*$ such that $L_{y,\ell} b L_{y,r} \subseteq L_y$.*

The proof follows from simple observations applied to Lemma 5.3: for appropriate m, if a regular language has an infinite intersection with $(a^m)^*$ ($(a^m)^* b (a^m)^*$) then it is in fact co-finite in it (for subsets of a^*ba^* please consult the statement).

To prove Theorem 5.1 we apply Lemma 5.4: given a sequent with regular constraints e_x, e_y on x, y we choose appropriate infinite sequences in the regular languages $L(e_x) \cap (a^m)^*, L(e_y) \cap (a^m)^* b (a^m)^*$, which is possible by the assumption that they satisfy condition (1), and show that after a rule application at least one of the sequents has regular constraints e'_x, e'_y, which have infinitely many elements from the sequences, and so by Lemma 5.4 they in fact satisfy condition (1). The choice of sequences depends on the applied rule; say, for a Cut rule, in which we replace the

regular constraint e_y with e_y, e and $e_y, \overline{e}$ observe that infinitely many elements of the sequence in $L(e_y)$ are in at least one of $L(e_y) \cap L(e)$ or $L(e_y) \cap \overline{L(e)}$.

6 Experiments

For evaluation, we have implemented a new string solver, RCP, which is built on top of OSTRICH. OSTRICH relies on a robust foundation—including the BRICS automata framework [Møller 2021], transducer support, and pre-implemented backward propagation—that enabled us to straightforwardly extend the solver with RCP.

To isolate the impact of RCP, we implemented it on top of the *basic OSTRICH* configuration, which combines backward propagation with equation splitting. In doing so, we disabled other advanced techniques used in OSTRICH's SMT-COMP portfolio, such as cost-enriched automata (CEA) [Chen et al. 2020] and the algebraic data type (ADT)-based solver [Hojjat and Rümmer 2017]. As a result, our evaluation considers two configurations of OSTRICH: (1) the *SMT-COMP portfolio version*, which integrates basic OSTRICH, CEA-OSTRICH, and the ADT-based solver; and (2) the *basic OSTRICH*, on which we directly implement and evaluate RCP.

This setup allows us to measure the isolated benefits of regular constraint propagation and compare it both to the unmodified basic configuration and the full SMT-COMP portfolio variant.

We evaluate RCP, as well as cvc5 [Barbosa et al. 2022], OSTRICH [Chen et al. 2023b], Z3 [de Moura and Bjørner 2008], Z3-alpha [Lu et al. 2024], and Z3-Noodler [Chen et al. 2023a], using their latest versions configured as in SMT-COMP 2024 [SMT 2024]. The experiments were conducted on a MacBook Pro with 16 GB of RAM, running macOS Sonoma 14.5. The system was powered by an Apple M3 chip. The timeout for each experiment was set to 60 seconds.

6.1 Proof Search

To transform the proof system (Figure 2) into a practical algorithm, one must define a strategy for applying the various proof rules. In principle, this can be done using Algorithm 1 for the case of orderable string constraints, as the algorithm outputs a sequence of propagation rules that can be executed to prove (un)satisfiability. While this provides a complete propagation order for formulas within the orderable fragment, it does not generalize to formulas outside this fragment. Moreover, our goal is to define a uniform procedure that applies broadly, even when completeness cannot be guaranteed.

As a baseline, we describe a simple "fair" scheduling strategy (for details see full version) that is easy to implement and ensures that every applicable propagation rule is eventually considered. To improve on this baseline, we employ a priority-based refinement of fair scheduling that guides propagation using a cost heuristic. Our solver introduces a bias toward inexpensive and potentially informative propagation steps, e.g., forward propagation from small automata or from functions with constant arguments while at the same time delaying applications that involve large automata or only yield coarse approximations. This heuristic assigns priorities to applications of forward/backward propagation using a weighted sum of factors, including:

- **Concrete arguments:** high priority is given to rules whose input or output language is a ground string;
- **Information gain:** backward rules propagating from universal automata (e.g., $x \in \Sigma^*$) receive lower priority;
- **Exactness:** forward rules involving functions without exact post-images (e.g., `replaceAll` with symbolic patterns) are penalized;
- **Cost:** rules are penalized in proportion to the combined size of input and result automata;

- **Fairness:** newer rule applications are penalized, preferring those that have waited longer in the queue.

6.2 Benchmarks

We evaluate our solver on three benchmark sets. The first benchmark set consists of 2,000 formulas randomly selected from the QF_S and QF_SLIA divisions of SMT-LIB 2024 [Preiner et al. 2024]. To ensure representativeness, we sampled proportionally to the distribution of benchmarks in the full SMT-LIB 2024 dataset, which contains approximately 100,000 benchmarks. This resulted in a subset of roughly 400 benchmarks from QF_S and 1,600 from QF_SLIA, preserving the overall ratio between the divisions.

Below, we provide descriptions for some of the different benchmark categories within the SMT-LIB:

- **Regular Constraints + Word Equations**: Benchmarks from *SyGuS*, *Automatark* [D'Antoni 2018], *Woorpje-lu* [Day et al. 2019], *String Fuzz* [Blotsky et al. 2018], *ReDoS Attack* [Li et al. 2021] and *Kepler* [Le and He 2018], which combine word equations with regular constraints and length.
- **String Constraints from Programs**: Extracted from concolic execution engines *Conpy* [Chen et al. 2014], symbolic execution of Python programs[Reynolds et al. 2017], *PyExZ3* [Ball and Daniel 2015], covering operations such as `replace`, `indexof`, `substr`.
- **String Rewrite**: Consists of benchmarks derived from axioms over string predicates and functions [Nötzli et al. 2019].
- **WebApp**: A variation of *SLOG* [Wang et al. 2016] using `replaceAll` instead of simple `replace`, derived from real web applications.

A key limitation of the SMT-LIB benchmark set is the relative scarcity of benchmarks containing *complex string functions*. While modern programming languages such as *JavaScript* heavily rely on operations like `replaceAll`, such functions are significantly underrepresented in the SMT-LIB benchmark releases. For example, only *0.4%* of the benchmarks in the SMT-LIB 2024 release include `replaceAll`.

The second benchmark set consists of instances of **PCP**, specifically in the form PCP[3,3], meaning each instance contains three dominos, and each word on a domino has length three. We generate 1,000 random instances over a binary alphabet, following the structure outlined in Example 2.2.

The third benchmark set is inspired by **bioinformatics** [Compeau and Pevzner 2015; Mount 2004] and models a reverse transcription process. In these benchmarks, an unknown RNA string y is transformed into a DNA string by applying a sequence of `replaceAll` operations that simulate nucleotide base pairing. The solver must determine an RNA string such that, after these replacement operations, a given concrete DNA string is produced, while ensuring that the RNA string contains a specific pattern. We generate 1000 instances, with 500 instances being satisfiable (where a suitable y exists) and 500 unsatisfiable. Each formula contains 4 `replaceAll` constraints, a constant DNA string of length 200, and a pattern string of length 15.

We also evaluated solver performance on the three illustrative formulas from Section 2: the string sanitization example, a simplified PCP tile instance, and a representative example from the bioinformatics benchmark set. Table 1 summarizes the results. RCP successfully solved all three instances, while existing solvers either timed out or returned `unknown`. The PCP formula is unsatisfiable, and only RCP was able to prove this. The normalization task lies outside the straight-line fragment and involves multiple transductions on the same variable; while OSTRICH supports

Table 1. Solver results on the motivating examples in Section 2. ✓= solved, ✗= timeout or unknown.

Solver	String Sanitization	PCP	Bioinformatics
RCP	✓	✓	✓
cvc5	✗	✗	✗
OSTRICH	✗	✗	✓
Z3	✗	✗	✗
Z3-alpha	✗	✗	✗
Z3-Noodler	✗	✗	✗

Table 2. Comparison of various solvers' performance on 2000 SMT-LIB'24 benchmarks, 1000 PCP benchmarks, and 1000 Bioinformatics benchmarks. For SMT-LIB'24, we report the number of satisfiable (Sat), unsatisfiable (Unsat), and unknown (Unknown) instances, along with total solving time (Time(s)), including timeouts. For PCP and Bioinformatics, we report the number of solved instances and total solving time. For all benchmarks, we conservatively assign a time of 60 seconds to each unknown result, even if the solver terminated earlier.

	SMT-LIB'24				**PCP**		**Bioinformatics**	
	Sat	Unsat	Unknown	Time(s)	Solved	Time(s)	Solved	Time(s)
RCP	1071	728	201	15416.8	901	7308.7	1000	5532.6
cvc5	1162	716	122	7954.3	0	60000.0	0	-
OSTRICH-BASE	833	653	514	32298.7	0	60000.0	1000	2596.2
OSTRICH-COMP	1009	733	258	22840.7	0	60000.0	1000	3911.0
Z3	1156	730	114	8286.0	0	60000.0	0	60000.0
Z3-alpha	1127	724	149	10681.2	0	60000.0	0	60000.0
Z3-Noodler	**1236**	**749**	**15**	797.0	0	60000.0	0	60000.0

transducers, it fails to solve this instance. By contrast, the bioinformatics example *is* within the straight-line fragment and can be solved by both RCP and OSTRICH, but not by any other solvers.

6.3 Performance of Individual Solvers

We begin by analyzing Table 2, which presents the performance of all solvers across both benchmark sets. We report results using different metrics for the two benchmark sets. We focus on *unsolved benchmarks* to highlight solver weaknesses.

For the first benchmark set, Z3-Noodler emerges as the best-performing solver, solving nearly all instances and leaving only ~*1%* of the benchmarks unsolved. It is closely followed by Z3 and cvc5, both of which leave *6%* of the benchmarks unsolved. Next, Z3-alpha performs slightly worse, failing to solve *7.5%* of the benchmarks. RCP leaves *10%* of the benchmarks unsolved. OSTRICH-COMP fails to solve *13%* of the benchmarks, while OSTRICH fails on *25%*.

These results demonstrate that while RCP does not achieve top-tier performance, it remains competitive using simple strategies, effectively solving a significant portion of the benchmark set. Notably, it reduces the number of unsolved benchmarks from *25%* (roughly 500) with *OSTRICH* to *10%* (around 200), a **60% reduction** in the number of failures.

A different trend emerges in the second and third benchmark sets, which consist of *1,000 PCP benchmarks* and *1,000 bioinformatics benchmarks*. In contrast to the SMT-LIB'24 benchmarks, most solvers fail to solve any instances from these categories. RCP is the only exception, solving nearly all PCP instances (with only around *10%* remaining unsolved). While OSTRICH is unable to solve the

Table 3. Benchmark Results for Different Solver Combinations. For each benchmark suite, we report the number of benchmarks left unknown by the portfolio (U), the additional benchmarks solved by RCP (C), and the resulting Improvement (I), defined as $\frac{C}{C+U}$. The last four rows below show additional RCP contribution when added to solvers already combined with OSTRICH-BASE (OB).

	SMT-LIB'24 (2000)			**PCP (1000)**			**Bioinformatics (1000)**		
Solver Combination	U	C	I	U	C	I	U	C	I
RCP	201	–	–	99	–	–	0	–	–
cvc5 + RCP	61	61	0.50	99	901	0.90	0	1000	1.00
OSTRICH-COMP + RCP	68	197	0.74	99	901	0.90	0	0	0.00
Z3 + RCP	85	29	0.25	99	901	0.90	0	1000	1.00
Z3-alpha + RCP	87	65	0.43	99	901	0.90	0	1000	1.00
Z3-Noodler + RCP	**11**	4	0.27	99	901	0.90	0	1000	1.00
(cvc5 + OB) + RCP	61	2	0.03	99	901	0.90	0	0	0.00
(Z3 + OB) + RCP	85	5	0.06	99	901	0.90	0	0	0.00
(Z3-alpha + OB) + RCP	87	8	0.08	99	901	0.90	0	0	0.00
(Z3-Noodler + OB) + RCP	**4**	0	0.00	99	901	0.90	0	0	0.00

PCP benchmarks, it manages to handle the bioinformatics benchmarks. This disparity for OSTRICH can be attributed to the underlying structure of the benchmarks: while the PCP benchmarks involve a more complex structure, the bioinformatics benchmarks fall into the straight-line fragment, for which OSTRICH is effective. These results highlight that outside of the SMT-LIB benchmark sets, many challenging benchmarks remain unsolved by existing solvers. However, our propagation-based approach solves a significant portion of these previously unsolved benchmarks, demonstrating that constraint propagation can be crucial for tackling problems beyond the reach of current solvers.

6.4 Impact of Enhancing Solver Performance with RCP

Table 3 presents the performance of RCP and its impact when combined with various solvers. Specifically, for each solver, we compute the *virtual best portfolio* (VBP), which considers the union of benchmarks solved either by the solver alone or by RCP. This is a particularly useful measure of impact, as many practical approaches for solving string constraints rely on a portfolio of solvers running in parallel. Evaluating the VBP provides insight into whether including a given solver in such a portfolio is beneficial, thereby helping to assess the practical value of RCP when integrated into a multi-solver strategy. We omit *OSTRICH-BASE* as base solver from this evaluation, as its technique is very close to RCP, and the previous experiments have already demonstrated the superior performance of RCP over this baseline.

We assess the unique contribution of RCP, defined as the number of benchmarks solved by RCP that the individual solver could not solve on its own. For each benchmark suite, we report three key quantities: the number of benchmarks left unsolved by the portfolio solver, the contribution provided by RCP, and the resulting improvement. The improvement is computed as $(\frac{\text{contribution}}{\text{unsolved+contribution}})$ which quantifies the relative gain achieved by including RCP in a portfolio. This metric effectively demonstrates the impact of RCP in complementing existing solvers.

First, it is immediately apparent that for the PCP benchmarks, no other solver is able to solve any instances, making RCP solely responsible for a contribution of 901 solved benchmarks in this set. As a result, all solver combinations with RCP show an improvement of approximately 90% on the PCP benchmarks. In addition, for the bioinformatics benchmarks the improvement is 100%

for all solver combinations except for OSTRICH + RCP, as OSTRICH can already solve the entire benchmark suite (yielding an improvement of 0%).

For the SMT-LIB benchmark set, the contribution of RCP varies noticeably across the different base solvers. For instance, when combined with cvc5, RCP contributes an additional 61 solved instances from 122 unknown benchmarks, resulting in an improvement of 50%. Meanwhile, combinations with Z3 and Z3-alpha show intermediate improvements of roughly 25% and 43%, respectively. These improvements follow the same pattern, primarily benefiting from enhanced handling of regular constraints, word equations, and length reasoning across various benchmark categories. In contrast, the combination of OSTRICH with RCP yields a significantly higher contribution, solving an extra 197 out of 258 unknown cases—translating to an improvement of about 74%.[3]

The improvements are most pronounced in the String Constraints from Programs and String Rewrite benchmarks. Its propagation strategy proves less effective on benchmarks involving more complex string functions and predicates, leading to weaker performance in those cases. Finally, Z3-Noodler sees a more modest contribution (4 additional instances from 11 unknown, or 27% improvement). This outcome is unsurprising, as Z3-Noodler also employs RCP as part of its strategy and solves already most of the benchmarks in this suite. The remaining unsolved benchmarks fall into two categories: *one* originates from String Constraints from Programs, while *three* belong to the WebApp category, which relies on `replaceAll`. Since this operation is currently unsupported in Z3-Noodler, these results indicate that while its propagation techniques are effective, its coverage of string functions in the SMT-LIB 2.7 format remains incomplete.

Notably, while Z3-Noodler showed only a minor absolute improvement on the SMT-LIB benchmarks, this can largely be attributed to the *underrepresentation of complex string functions* such as `replaceAll` in the SMT-LIB dataset. Since Z3-Noodler already excels at solving the predominant types of string constraints in SMT-LIB, its gains from RCP are limited. However, as observed in the WebApp benchmarks, where `replaceAll` is more frequently used, RCP provided improvements.

The last four rows of Table 3 show the additional contribution of RCP when added to portfolios already containing OSTRICH-BASE. As expected, the improvements on SMT-LIB'24 benchmarks are marginal (ranging from 0% to 8%), due to significant overlap between the constraints solved by RCP and OSTRICH-BASE. This highlights a natural trend: as more solvers are combined in a portfolio, the marginal contribution of each individual technique diminishes. Nonetheless, the value of RCP becomes more apparent on harder or previously unaddressed benchmarks. In particular, while OSTRICH-BASE fails to solve any PCP instances, adding RCP recovers 901 solved cases across all configurations, demonstrating its unique strength in handling expressive string constraints beyond SMT-LIB.

7 Related Work

String solving has been extensively studied in the past decade or so, e.g., see the survey [Amadini 2023]. Our work contributes to both the theoretical and practical aspects of string solving. In this section, we organize the related literature into two complementary parts. First, we review the theoretical foundations that underpin string constraint solving—ranging from early results on decidability. Second, we discuss practical string solvers, highlighting the evolution of implementations from early prototypes to state-of-the-art tools.

Theoretical Foundations. Early work on the satisfiability of word equations established decidability with significant complexity results. Makanin [Makanin 1977] and later Plandowski [Plandowski

[3]This also provides a strong indication of the potential improvement in *OSTRICH-COMP* if the *OSTRICH-BASE* component were replaced by our RCP implementation—reducing the number of unsolved instances to just 3%, and ranking just behind *Z3-Noodler*.

1999] showed that the problem is decidable and lies in PSPACE, while subsequent work by Jeż [Jeż 2016] introduced the recompression technique—a simpler method that streamlined earlier approaches. In practice, Nielsen's transformation [Nielsen 1917] has emerged as an effective technique for handling quadratic word equations, maintaining the same theoretical complexity bounds. Although the lower bound is known to be NP, establishing tight bounds remains challenging.

The decidability of word equations with linear length constraints such as $|x| = |y|$ remains an open problem. However, for quadratic word equations, some positive results have been obtained by combining Nielsen's transformation with counters to track variable lengths [Lin and Majumdar 2021]. Moreover, even simple extensions incorporating transducers render the problem undecidable, which motivated the definition of the straight-line fragment [Lin and Barceló 2016]. This fragment restricts constraints so that each variable is assigned at most once—an intuitively appealing condition that nonetheless captures a rich class of problems. While the computational complexity of the straight-line fragment is EXPSPACE in general, it drops to PSPACE for constraints of small dimensions (i.e., when the structure of the constraints is sufficiently limited) and remains decidable even when augmented with additional constraints such as length, letter counting, and disequality. Recent work has further refined these ideas: a comprehensive proof system for string constraints that captures the straight-line fragment was introduced in [Chen et al. 2022], providing a formal framework that informs our approach, and the straight-line condition was extended to the theory of sequences in [Jeż et al. 2023] by applying constraint propagation techniques on parametric languages rather than on regular languages (following a similar approach to that in [Chen et al. 2019]).

Building on the idea of limiting variable assignments to ensure decidability, the chain-free fragment [Abdulla et al. 2019] extends the decidable class of string constraints by allowing variables to be assigned more than once, provided that these assignments do not form a chain. Although this extension builds on the concepts underlying the straight-line fragment, the techniques employed are fundamentally different. In the chain-free fragment, a splitting graph is introduced to systematically manage the interactions between variables—particularly when a variable appears multiple times on the same side of an equation—thus enabling a richer set of constraints while preserving decidability.

In this context, our work shows that the proof system from [Chen et al. 2022], specifically the rules for regular constraint propagation, is sufficient to subsume the chain-free fragment, providing a novel algorithmic characterization of it.

String Solvers. Over the past decade, a wide range of string solvers have emerged, employing techniques such as reductions to other theories, word equation splitting, and automata-based reasoning. Many automata-based solvers apply constraint propagation, typically using limited forms of automata splitting over concatenation.

Kaluza [Saxena et al. 2010] and HAMPI [Kiezun et al. 2013] handle string constraints over bounded-length variables by translating them into bit-vector constraints. Similarly, G-Strings [Amadini et al. 2017] and Gecode+s [Scott et al. 2017] model strings as arrays of integers along with explicit length constraints, relying on constraint propagation in those domains. nfa2sat [Lotz et al. 2023] and Woorpje [Day et al. 2019] encode string problems into propositional logic and solve them via SAT solvers, although Woorpje is limited to bounded-length word equations while nfa2sat supports more general cases. Sloth [Holík et al. 2017], Qzy [Cox and Leasure 2017], and SLOG/Slent [Wang et al. 2018, 2016] use automata to represent string constraints and reduce the problem to a model checking task—often solved using tools like IC3 [Bradley 2011]. PISA [Tateishi et al. 2013], on the other hand, maps string constraints to monadic second-order logic on finite strings handling `indexOf` and `replace` operations.

The Z3Str family—comprising Z3Str/2/3/4 [Berzish et al. 2017; Mora et al. 2021; Zheng et al. 2015, 2013]—iteratively decomposes constant strings into substrings and splits variables into subvariables until they become bounded. Z3str2 introduces new search heuristics and detects overlapping variables, while Z3str3 extends this with theory-aware branching to prioritize easier constraints. Z3STR3RE [Berzish et al. 2023, 2021] further refines the approach by handling regular constraints directly without reducing them to word equations. Building on these ideas, Z3-alpha [Lu et al. 2024, 2023] employs an SMT strategy synthesis technique to select appropriate solving strategies based on the underlying solver (e.g., Z3 or Z3Str4). Z3 [de Moura and Bjørner 2008] integrates multiple techniques—including those from Z3-alpha and Z3seq [Stanford et al. 2021], which is based on symbolic automata and derivatives—to capture the theory of sequences. While many state-of-the-art string solvers use rewriting as a preprocessing step or to handle specific string functions, cvc5 [Barbosa et al. 2022] distinguishes itself by relying on derivation rules as a core strategy, lazily reducing complex string functions to word equations.

Other solvers focus on automata-based techniques and constraint propagation. Early ideas[Golden and Pang 2003] applied simplistic regular constraint propagation, while later work [Minamide 2005] introduced forward propagation for web page analysis, supporting complex string functions and transductions. STRANGER [Yu et al. 2010] uses both forward and backward propagation to over-approximate the inputs of string variables at various program points. Norn [Abdulla et al. 2015, 2014] guarantees termination on the acyclic fragment through automata splitting and length propagation.

Trau [Abdulla et al. 2021, 2018, 2017] extends Norn's approach with counterexample-guided abstraction refinement to approximate regular languages using arithmetic formulas and is the first solver to handle the chain-free fragment [Abdulla et al. 2019], supporting features such as string transductions and `replaceAll`. CertiStr [Kan et al. 2022] adopts a forward propagation strategy over regular constraints and concatenation. OSTRICH [Chen et al. 2019] employs refined backward propagation—along with word equation splitting, length abstraction, and letter counting—to propagate regular constraints across string functions (including `replaceAll` and transductions), thereby guaranteeing completeness for the straight-line fragment. In practice, OSTRICH is run at SMT-COMP using a portfolio configuration with timeslicing, where three different base solvers are executed sequentially within a fixed time budget: the basic OSTRICH (which combines backward propagation with equation splitting), CEA-OSTRICH [Chen et al. 2020] (which uses cost-enriched automata for improved reasoning about length, `indexOf`, and `subString` constraints), and a solver based on algebraic data types [Hojjat and Rümmer 2017]. Z3-Noodler [Blahoudek et al. 2023; Chen et al. 2023a; Havlena et al. 2024] introduces a specialized stabilization algorithm, applying a combination of forward and backward propagation to a selected word equation until the inferred regular languages on both sides stabilize. Its techniques have been extended to handle length constraints, disequalities, and string-to-integer conversion. Although complete for the chain-free fragment (on supported string functions), Z3-Noodler does not support string transductions and `replaceAll`—a gap our solver addresses.

Owing to the interest in and the demands of string solving, a standard file format has been recently agreed and adopted as part of SMT-LIB 2.6 [Barrett et al. 2024], called the theory of Unicode strings (introduced in 2020). This has made it possible to create a track on string problems at the SMT-COMP [Barrett et al. 2005].[4]

[4]Not all aforementioned solvers support SMT-LIB 2.6 format.

8 Conclusion and Future Work

In this paper, we demonstrated that a simple strategy like RCP can effectively solve a wide range of benchmarks in string solving. We first established its completeness on the orderable fragment, which subsumes two of the largest known decidable fragments: *chain-free* and *straight-line* constraints.

We introduced a fair proof search strategy for applying RCP, ensuring that every constraint is eventually propagated and no propagation rule is indefinitely postponed. However, the efficiency of proof search may vary depending on the structure of the input constraints. For instance, the Marking Algorithm provides a specialized strategy that is complete for the orderable fragment but does not generalize beyond it. Furthermore, our strategies are not without limitations. They are not complete for certain simple equations, and our exploration has not utilized the entire proof system. There are rules that are theoretically not fully understood yet, such as the "Cut" rule, which allows splitting the search around arbitrary regular constraints. While this rule can be advantageous in many cases, its integration poses a challenge as it is not clear how to algorithmically choose a constraint to split around. Developing adaptive heuristics for proof search, tailored to specific classes of string constraints, remains an interesting direction for future work.

Beyond our theoretical contributions, we implemented RCP based on our proof system and evaluated its performance on standard SMT string benchmarks. Although RCP does not outperform state-of-the-art solvers on the overall SMT-COMP datasets, it remains competitive, solving up to 90% of the benchmarks—compared to other solvers that solve between 75% and 99% of the set. Notably, our work significantly improves the performance of base OSTRICH, increasing its benchmark-solving ratio from 75% to 90%. When combined with the remainder of the OSTRICH portfolio, the resulting virtual best portfolio achieves up to 97% of solved benchmarks—second only to Z3-Noodler on the SMT-COMP benchmarks. Moreover, RCP excels in previously unsolved benchmark categories, namely on PCP instances and on bioinformatics benchmarks, where only OSTRICH achieves comparable results. Finally, integrating RCP with existing solvers significantly enhances their performance, reducing the number of unsolved benchmarks by 25% to 74% on SMT-COMP datasets and by up to 100% on both PCP and bioinformatics benchmarks.

While our work primarily focuses on string constraints over regular languages, an interesting direction for future research is to explore the applicability of our techniques in the theory of sequences, where the underlying languages are specified by parametric automata. As noted in the related work, preliminary investigations using constraint propagation methods in this domain have yielded promising results. Further exploration could provide valuable insights into both the theoretical and practical implications of extending our approach to sequence constraints.

Acknowledgments

We thank anonymous reviewers for their valuable feedback. This work was supported by the Engineering and Physical Sciences Research Council (EPSRC) under Grant No. EP/T00021X/1; the European Research Council (ERC) under Grant No. 101089343 (LASD); and the Swedish Research Council under Grant No. 2021-06327. For the purpose of open access, the author has applied a Creative Commons Attribution (CC BY) licence to any Author Accepted Manuscript version arising from this submission.

Data Availability Statement

The complete implementation of RCP, along with the benchmark suites (including the SMT-LIB, PCP, and bioinformatics benchmarks) are available on Zenodo [Markgraf et al. 2025].

References

2013. OWASP: Top10. (2013). https://owasp.org/www-pdf-archive/OWASP_Top_10_-_2013.pdf

2017. OWASP: Top10. (2017). https://owasp.org/www-project-top-ten/2017/

2021. OWASP: Top10. (2021). https://owasp.org/Top10/

2024. SMT-COMP 2024: The 19th International Satisfiability Modulo Theories Competition. https://smt-comp.github.io/2024/.

Parosh Aziz Abdulla, Mohamed Faouzi Atig, Yu-Fang Chen, Bui Phi Diep, Lukáš Holík, Denghang Hu, Wei-Lun Tsai, Zhillin Wu, and Di-De Yen. 2021. Solving Not-Substring Constraint with Flat Abstraction. In *Programming Languages and Systems*, Hakjoo Oh (Ed.). Springer International Publishing, Cham, 305–320. doi:10.1007/978-3-030-89051-3_17

Parosh Aziz Abdulla, Mohamed Faouzi Atig, Yu-Fang Chen, Bui Phi Diep, Lukas Holik, Ahmed Rezine, and Philipp Ruemmer. 2018. Trau: SMT solver for string constraints. *2018 Formal Methods in Computer Aided Design (FMCAD)* (2018), 1–5. doi:10.23919/FMCAD.2018.8602997

Parosh Aziz Abdulla, Mohamed Faouzi Atig, Yu-Fang Chen, Bui Phi Diep, Lukáš Holík, Ahmed Rezine, and Philipp Rümmer. 2017. Flatten and conquer: a framework for efficient analysis of string constraints. *SIGPLAN Not.* 52, 6 (June 2017), 602–617. doi:10.1145/3140587.3062384

Parosh Aziz Abdulla, Mohamed Faouzi Atig, Yu-Fang Chen, Lukáš Holík, Ahmed Rezine, Philipp Ruemmer, and Jari Stenman. 2015. Norn: An SMT Solver for String Constraints. In *Computer Aided Verification*, Daniel Kroening and Corina S. Păsăreanu (Eds.). Springer International Publishing, Cham, 462–469. doi:10.1007/978-3-319-21690-4_29

Parosh Aziz Abdulla, Mohamed Faouzi Atig, Yu-Fang Chen, Lukáš Holík, Ahmed Rezine, Philipp Rümmer, and Jari Stenman. 2014. String Constraints for Verification. In *Computer Aided Verification*, Armin Biere and Roderick Bloem (Eds.). Springer International Publishing, Cham, 150–166. doi:10.1007/978-3-319-08867-9_10

Parosh Aziz Abdulla, Mohamed Faouzi Atig, Bui Phi Diep, Lukáš Holík, and Petr Janků. 2019. Chain-Free String Constraints. In *Automated Technology for Verification and Analysis*, Yu-Fang Chen, Chih-Hong Cheng, and Javier Esparza (Eds.). Springer International Publishing, Cham, 277–293. doi:10.1007/978-3-030-31784-3_16

Rajeev Alur and Pavol Cerný. 2010. Expressiveness of streaming string transducers. In *IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science, FSTTCS 2010, December 15-18, 2010, Chennai, India.* 1–12. doi:10.4230/LIPIcs.FSTTCS.2010.1

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.* 1–20. doi:10.1007/978-3-642-22012-8_1

Roberto Amadini. 2023. A Survey on String Constraint Solving. *ACM Comput. Surv.* 55, 2 (2023), 16:1–16:38. doi:10.1145/3484198

Roberto Amadini, Graeme Gange, Peter J. Stuckey, and Guido Tack. 2017. A Novel Approach to String Constraint Solving. In *Principles and Practice of Constraint Programming*, J. Christopher Beck (Ed.). Springer International Publishing, Cham, 3–20. doi:10.1007/978-3-319-66158-2_1

Thomas Ball and Jakub Daniel. 2015. Deconstructing Dynamic Symbolic Execution. In *Dependable Software Systems Engineering*, Maximilian Irlbeck, Doron A. Peled, and Alexander Pretschner (Eds.). NATO Science for Peace and Security Series, D: Information and Communication Security, Vol. 40. IOS Press, 26–41. doi:10.3233/978-1-61499-495-4-26

Haniel Barbosa, Clark Barrett, Martin Brain, Gereon Kremer, Hanna Lachnitt, Makai Mann, Abdalrhman Mohamed, Mudathir Mohamed, Aina Niemetz, Andres Nötzli, Alex Ozdemir, Mathias Preiner, Andrew Reynolds, Ying Sheng, Cesare Tinelli, and Yoni Zohar. 2022. *cvc5: A Versatile and Industrial-Strength SMT Solver.* 415–442. doi:10.1007/978-3-030-99524-9_24

Clark Barrett, Leonardo de Moura, and Aaron Stump. 2005. SMT-COMP: Satisfiability Modulo Theories Competition. In *Computer Aided Verification*, Kousha Etessami and Sriram K. Rajamani (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 20–23. doi:10.1007/11513988_4

Clark Barrett, Pascal Fontaine, and Cesare Tinelli. 2024. The SMT-LIB Standard: Version 2.6. http://smt-lib.org.

Jean Berstel. 1979. *Transductions and Context-Free Languages.* Teubner-Verlag. doi:10.1007/978-3-663-09367-1

Murphy Berzish, Joel D. Day, Vijay Ganesh, Mitja Kulczynski, Florin Manea, Federico Mora, and Dirk Nowotka. 2023. Towards more efficient methods for solving regular-expression heavy string constraints. *Theoretical Computer Science* 943 (2023), 50–72. doi:10.1016/j.tcs.2022.12.009

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).* 55–59. doi:10.23919/FMCAD.2017.8102241

Murphy Berzish, Mitja Kulczynski, Federico Mora, Florin Manea, Joel D. Day, Dirk Nowotka, and Vijay Ganesh. 2021. An SMT Solver for Regular Expressions and Linear Arithmetic over String Length. In *Computer Aided Verification*, Alexandra Silva and K. Rustan M. Leino (Eds.). Springer International Publishing, Cham, 289–312. doi:10.1007/978-3-030-81688-9_14

František Blahoudek, Yu-Fang Chen, David Chocholatý, Vojtěch Havlena, Lukáš Holík, Ondřej Lengál, and Juraj Síč. 2023. Word Equations in Synergy with Regular Constraints. In *Formal Methods*, Marsha Chechik, Joost-Pieter Katoen, and Martin Leucker (Eds.). Springer International Publishing, Cham, 403–423. doi:10.1007/978-3-031-27481-7_23

Dmitry Blotsky, Federico Mora, Murphy Berzish, Yunhui Zheng, Ifaz Kabir, and Vijay Ganesh. 2018. StringFuzz: A Fuzzer for String Solvers. In *Computer Aided Verification*, Hana Chockler and Georg Weissenbacher (Eds.). Springer International Publishing, Cham, 45–51. doi:10.1007/978-3-319-96142-2_6

Aaron R. Bradley. 2011. SAT-based model checking without unrolling. In *International Conference on Computer Aided Verification (CAV)*. Springer, 70–87. doi:10.1007/978-3-642-18275-4_7

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

Taolue Chen, Alejandro Flores-Lamas, Matthew Hague, Zhilei Han, Denghang Hu, Shuanglong Kan, Anthony W. Lin, Philipp Ruemmer, and Zhilin Wu. 2022. Solving string constraints with Regex-dependent functions through transducers with priorities and variables. *Proc. ACM Program. Lang.* 6, POPL (2022), 1–31. doi:10.1145/3498707

Taolue Chen, Matthew Hague, Jinlong He, Denghang Hu, Anthony Widjaja Lin, Philipp Rümmer, and Zhilin Wu. 2020. A Decision Procedure for Path Feasibility of String Manipulating Programs with Integer Data Type. In *Automated Technology for Verification and Analysis*, Dang Van Hung and Oleg Sokolsky (Eds.). Springer International Publishing, Cham, 325–342. doi:10.1007/978-3-030-59152-6_18

Taolue Chen, Matthew Hague, Anthony Lin, Philipp Ruemmer, and Zhilin Wu. 2019. Decision procedures for path feasibility of string-manipulating programs with complex operations. *Proceedings of the ACM on Programming Languages* 3 (01 2019), 1–30. doi:10.1145/3290362

Taolue Chen, Riccardo De Masellis, Alejandro Flores-Lamas, Matthew Hague, Zhilei Han, Denghang Hu, Oliver Markgraf, Anthony W. Lin, Philipp Ruemmer, and Zhilin Wu. 2023b. OSTRICH Version 1.3. (2023). https://www.cs.rhul.ac.uk/home/uxac009/files/papers/smtcomp2023.pdf

Ting Chen, Xiao-song Zhang, Rui-dong Chen, Bo Yang, and Yang Bai. 2014. Conpy: Concolic Execution Engine for Python Applications. In *Algorithms and Architectures for Parallel Processing*, Xian-he Sun, Wenyu Qu, Ivan Stojmenovic, Wanlei Zhou, Zhiyang Li, Hua Guo, Geyong Min, Tingting Yang, Yulei Wu, and Lei Liu (Eds.). Springer International Publishing, Cham, 150–163. doi:10.1007/978-3-319-11194-0_12

Yu-Fang Chen, David Chocholaty, Vojtech Havlena, Lukas Holik, Ondrej Lengal, and Juraj Sic. 2023a. Solving String Constraints with Lengths by Stabilization. *Proc. ACM Program. Lang.* 7, OOPSLA2, Article 296 (oct 2023), 30 pages. doi:10.1145/3622872

Yu-Fang Chen, David Chocholatý, Vojtěch Havlena, Lukáš Holík, Ondřej Lengál, and Juraj Síč. 2024. Z3-Noodler: An Automata-based String Solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, Bernd Finkbeiner and Laura Kovács (Eds.). Springer Nature Switzerland, Cham, 24–33. doi:10.1007/978-3-031-57246-3_2

Phillip Compeau and Pavel Pevzner. 2015. *Bioinformatics Algorithms: An Active Learning Approach, Volume 1* (2 ed.). Active Learning Publishers, La Jolla, CA.

Arlen Cox and Jason Leasure. 2017. Model Checking Regular Language Constraints. *CoRR* abs/1708.09073 (2017). arXiv:1708.09073 doi:10.48550/arXiv.1708.09073

L D'Antoni. 2018. Automatark automata benchmark (2018). https://github.com/lorisdanto/automatark

Joel D. Day, Thorsten Ehlers, Mitja Kulczynski, Florin Manea, Dirk Nowotka, and Danny Bøgsted Poulsen. 2019. On Solving Word Equations Using SAT. In *Reachability Problems*, Emmanuel Filiot, Raphaël Jungers, and Igor Potapov (Eds.). Springer International Publishing, Cham, 93–106. doi:10.1007/978-3-030-30806-3_8

Leonardo de Moura and Nikolaj Bjørner. 2008. Z3: an efficient SMT solver. *Tools and Algorithms for the Construction and Analysis of Systems* 4963, 337–340. doi:10.1007/978-3-540-78800-3_24

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. doi:10.1017/CBO9781107326019.013

G. Gentzen. 1935. Untersuchungen über das logische Schließen I. *Mathematische Zeitschrift* 39 (1935), 176–210. doi:10.1007/BF01201353

Keith Golden and Wanlin Pang. 2003. Constraint Reasoning over Strings. In *Principles and Practice of Constraint Programming – CP 2003*, Francesca Rossi (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 377–391. doi:10.1007/978-3-540-45193-8_26

Matthew Hague, Artur Jeż, Anthony W. Lin, Oliver Markgraf, and Philipp Rümmer. 2025. The Power of Regular Constraint Propagation (Technical Report). arXiv:2508.19888 [cs.LO] doi:10.48550/arXiv.2508.19888

Vojtěch Havlena, Lukáš Holík, Ondřej Lengál, and Juraj Síč. 2024. Cooking String-Integer Conversions with Noodles. In *27th International Conference on Theory and Applications of Satisfiability Testing (SAT 2024) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 305)*, Supratik Chakraborty and Jie-Hong Roland Jiang (Eds.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 14:1–14:19. doi:10.4230/LIPIcs.SAT.2024.14

Hossein Hojjat and Philipp Rümmer. 2017. Deciding and Interpolating Algebraic Data Types by Reduction. In *19th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing, SYNASC 2017, Timisoara, Romania, September 21-24, 2017*, Tudor Jebelean, Viorel Negru, Dana Petcu, Daniela Zaharie, Tetsuo Ida, and Stephen M. Watt (Eds.). IEEE Computer Society, 145–152. doi:10.1109/SYNASC.2017.00033

Lukáš Holík, Petr Janků, Anthony W. Lin, Philipp Rümmer, and Tomáš Vojnar. 2017. String constraints with concatenation and transducers solved efficiently. *Proc. ACM Program. Lang.* 2, POPL, Article 4 (Dec. 2017), 32 pages. doi:10.1145/3158092

Artur Jeż. 2016. Recompression: A Simple and Powerful Technique for Word Equations. *J. ACM* 63, 1, Article 4 (feb 2016), 51 pages. doi:10.1145/2743014

Artur Jeż, Anthony W. Lin, Oliver Markgraf, and Philipp Rümmer. 2023. Decision Procedures for Sequence Theories. In *Computer Aided Verification - 35th International Conference, CAV 2023, Paris, France, July 17-22, 2023, Proceedings, Part II (Lecture Notes in Computer Science, Vol. 13965)*, Constantin Enea and Akash Lal (Eds.). Springer, 18–40. doi:10.1007/978-3-031-37703-7_2

Shuanglong Kan, Anthony Widjaja Lin, Philipp Rümmer, and Micha Schrader. 2022. CertiStr: a certified string solver. In *Proceedings of the 11th ACM SIGPLAN International Conference on Certified Programs and Proofs* (Philadelphia, PA, USA) *(CPP 2022)*. Association for Computing Machinery, New York, NY, USA, 210–224. doi:10.1145/3497775.3503691

Adam Kiezun, Vijay Ganesh, Shay Artzi, Philip J. Guo, Pieter Hooimeijer, and Michael D. Ernst. 2013. HAMPI: A solver for word equations over strings, regular expressions, and context-free grammars. *ACM Trans. Softw. Eng. Methodol.* 21, 4, Article 25 (feb 2013), 28 pages. doi:10.1145/2377656.2377662

Quang Loc Le and Mengda He. 2018. A Decision Procedure for String Logic with Quadratic Equations, Regular Expressions and Length Constraints. In *Programming Languages and Systems*, Sukyoung Ryu (Ed.). Springer International Publishing, Cham, 350–372. doi:10.1007/978-3-030-02768-1_19

Yeting Li, Zixuan Chen, Jialun Cao, Zhiwu Xu, Qiancheng Peng, Haiming Chen, Liyuan Chen, and Shing-Chi Cheung. 2021. ReDoSHunter: A Combined Static and Dynamic Approach for Regular Expression DoS Detection. In *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, 3847–3864. https://www.usenix.org/conference/usenixsecurity21/presentation/li-yeting

Anthony Widjaja 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 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*. 123–136. doi:10.1145/2837614.2837641

Anthony W. Lin and Rupak Majumdar. 2021. Quadratic Word Equations with Length Constraints, Counter Systems, and Presburger Arithmetic with Divisibility. *Log. Methods Comput. Sci.* 17, 4 (2021). doi:10.46298/LMCS-17(4:4)2021

Kevin Lotz, Amit Goel, Bruno Dutertre, Benjamin Kiesl-Reiter, Soonho Kong, Rupak Majumdar, and Dirk Nowotka. 2023. Solving String Constraints Using SAT. In *Computer Aided Verification*, Constantin Enea and Akash Lal (Eds.). Springer Nature Switzerland, Cham, 187–208. doi:10.1007/978-3-031-37703-7_9

Zhengyang Lu, Stefan Siemer, Piyush Jha, Joel Day, Florin Manea, and Vijay Ganesh. 2024. Layered and Staged Monte Carlo Tree Search for SMT Strategy Synthesis. In *Proceedings of the Thirty-Third International Joint Conference on Artificial Intelligence, IJCAI-24*, Kate Larson (Ed.). International Joint Conferences on Artificial Intelligence Organization, 1907–1915. doi:10.24963/ijcai.2024/211 Main Track.

Zhengyang Lu, Stefan Siemer, Piyush Jha, Florin Manea, Joel Day, and Vijay Ganesh. 2023. Z3-alpha: A Reinforcement Learning Guided SMT Solver SMT-COMP 2023. (2023). https://smt-comp.github.io/2023/system-descriptions/z3-alpha.pdf

G. S. Makanin. 1977. The Problem of Solvability of Equations in a Free Semigroup. *Mathematics of The Ussr-sbornik* 32 (1977), 129–198. doi:10.1070/SM1977V032N02ABEH002376

Oliver Markgraf, Artur Jeż, Matthew Hague, Philipp Rümmer, and Anthony Widjaja Lin. 2025. The Power of Regular Constraint Propagation. Artifact on Zenodo. doi:10.5281/zenodo.15805177

Yasuhiko Minamide. 2005. Static approximation of dynamically generated Web pages. In *Proceedings of the 14th international conference on World Wide Web, WWW 2005, Chiba, Japan, May 10-14, 2005*, Allan Ellis and Tatsuya Hagino (Eds.). ACM, 432–441. doi:10.1145/1060745.1060809

Anders Møller. 2021. dk.brics.automaton – Finite-State Automata and Regular Expressions for Java. https://www.brics.dk/automaton.

Federico Mora, Murphy Berzish, Mitja Kulczynski, Dirk Nowotka, and Vijay Ganesh. 2021. Z3str4: A Multi-armed String Solver. In *Formal Methods: 24th International Symposium, FM 2021, Virtual Event, November 20–26, 2021, Proceedings*. Springer-Verlag, Berlin, Heidelberg, 389–406. doi:10.1007/978-3-030-90870-6_21

David W. Mount. 2004. *Bioinformatics: Sequence and Genome Analysis* (2 ed.). Cold Spring Harbor Laboratory Press, Cold Spring Harbor, NY.

Jakob Nielsen. 1917. Die Isomorphismen der allgemeinen, unendlichen Gruppe mit zwei Erzeugenden. *Math. Ann.* 78 (1917), 385–397. doi:10.1007/BF01457113

Andres Nötzli, Andrew Reynolds, Haniel Barbosa, Aina Niemetz, Mathias Preiner, Clark Barrett, and Cesare Tinelli. 2019. Syntax-Guided Rewrite Rule Enumeration for SMT Solvers. In *Theory and Applications of Satisfiability Testing – SAT 2019*, Mikoláš Janota and Inês Lynce (Eds.). Springer International Publishing, Cham, 279–297. doi:10.1007/978-3-030-24258-9_20

Wojciech Plandowski. 1999. Satisfiability of word equations with constants is in NEXPTIME. In *Proceedings of the Thirty-First Annual ACM Symposium on Theory of Computing* (Atlanta, Georgia, USA) *(STOC '99)*. Association for Computing Machinery, New York, NY, USA, 721–725. doi:10.1145/301250.301443

Mathias Preiner, Hans-Jörg Schurr, Clark Barrett, Pascal Fontaine, Aina Niemetz, and Cesare Tinelli. 2024. SMT-LIB release 2024 (non-incremental benchmarks). https://doi.org/10.5281/zenodo.11061097. doi:10.5281/zenodo.11061097 Version 2024.04.23.

Andrew Reynolds, Maverick Woo, Clark Barrett, David Brumley, Tianyi Liang, and Cesare Tinelli. 2017. Scaling Up DPLL(T) String Solvers Using Context-Dependent Simplification. In *Computer Aided Verification*, Rupak Majumdar and Viktor Kunčak (Eds.). Springer International Publishing, Cham, 453–474. doi:10.1007/978-3-319-63390-9_24

Francesca Rossi, Peter van Beek, and Toby Walsh. 2006. *Handbook of Constraint Programming*. Elsevier. doi:10.5555/2843512

Prateek Saxena, Devdatta Akhawe, Steve Hanna, Feng Mao, Stephen McCamant, and Dawn Song. 2010. A Symbolic Execution Framework for JavaScript. In *2010 IEEE Symposium on Security and Privacy*. 513–528. doi:10.1109/SP.2010.38

Joseph D. Scott, Pierre Flener, Justin Pearson, and Christian Schulte. 2017. Design and Implementation of Bounded-Length Sequence Variables. In *Integration of AI and OR Techniques in Constraint Programming*, Domenico Salvagnin and Michele Lombardi (Eds.). Springer International Publishing, Cham, 51–67. doi:10.1007/978-3-319-59776-8_5

Caleb Stanford, Margus Veanes, and Nikolaj Bjørner. 2021. Symbolic Boolean derivatives for efficiently solving extended regular expression constraints. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (Virtual, Canada) *(PLDI 2021)*. Association for Computing Machinery, New York, NY, USA, 620–635. doi:10.1145/3453483.3454066

Takaaki Tateishi, Marco Pistoia, and Omer Tripp. 2013. Path- and index-sensitive string analysis based on monadic second-order logic. *ACM Trans. Softw. Eng. Methodol.* 22, 4, Article 33 (oct 2013), 33 pages. doi:10.1145/2522920.2522926

Hung-En Wang, Shih-Yu Chen, Fang Yu, and Jie-Hong R. Jiang. 2018. A Symbolic Model Checking Approach to the Analysis of String and Length Constraints. In *2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 623–633. doi:10.1145/3238147.3238189

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*, Swarat Chaudhuri and Azadeh Farzan (Eds.). Springer International Publishing, Cham, 241–260. doi:10.1007/978-3-319-41528-4_13

Fang Yu, Muath Alkhalaf, and Tevfik Bultan. 2010. Stranger: An Automata-Based String Analysis Tool for PHP. 154–157. doi:10.1007/978-3-642-12002-2_13

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*, Daniel Kroening and Corina S. Păsăreanu (Eds.). Springer International Publishing, Cham, 235–254. doi: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 *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering* (Saint Petersburg, Russia) *(ESEC/FSE 2013)*. Association for Computing Machinery, New York, NY, USA, 114–124. doi:10.1145/2491411.2491456

Received 2025-03-26; accepted 2025-08-12