Formal Methods in Computer-Aided Design 2025

OSTRICH2: Solver for Complex String Constraints

Matthew Hague
Department of Computer Science
Royal Holloway, University of London
matthew.hague@rhul.ac.uk

Denghang Hu
Key Laboratory of System Software
Institute of Software
Chinese Academy of Sciences
hudh@ios.ac.cn

Artur Jeż
Institute of Computer Science
University of Wrocław
aje@cs.uni.wroc.pl

Anthony W. Lin
University of Kaiserslautern-Landau
Max-Planck Institute for Software Systems
awlin@mpi-sws.org

Oliver Markgraf
University of Kaiserslautern-Landau
markgraf@cs.uni-kl.de

Philipp Rümmer
University of Regensburg
Uppsala University
philipp.ruemmer@ur.de

Zhilin Wu
Key Laboratory of System Software
Institute of Software
Chinese Academy of Sciences
wuzl@ios.ac.cn

***Abstract*—We present OSTRICH2, the latest evolution of the SMT solver OSTRICH for string constraints. OSTRICH2 supports a wide range of complex functions on strings and provides completeness guarantees for a substantial fragment of string constraints, including the straight-line fragment and the chain-free fragment. OSTRICH2 provides full support for the SMT-LIB theory of Unicode strings, extending the standard with several unique features not found in other solvers: among others, parsing of ECMAScript regular expressions (including look-around assertions and capture groups) and handling of user-defined string transducers. We empirically demonstrate that OSTRICH2 is competitive to other string solvers on SMT-LIB benchmarks.**

I. Introduction

Strings are ubiquitous in modern software systems, especially with the advent of programming languages like JavaScript, PHP, and Python. Despite this, string manipulation is well-known to be error-prone and can easily lead to security vulnerabilities including HTML injection (e.g. see [1], [2]). Applications to analysis of security vulnerabilities caused by string manipulation have been one of the main catalysts for the extensive research into SMT over strings spanning across the last twenty years [2], [3], [4], [5], [6], [7], [8]. One of the success stories of string solvers includes their usage at AWS for analysis of Role-Based Access Control (RBAC) policies (e.g. see [9], [10]). Since 2020 SMT-LIB Unicode String theory [11] has been formalized and supported by many existing string solvers, including Z3 [12], Z3-alpha [13], Z3-Noodler [14], [15], [16], [17], Z3str3RE [18], cvc5 [19], Z3str4 [20], Trau [21], [22], [23], and our own solver OSTRICH [4].

In this paper, we present OSTRICH2, the latest evolution of the OSTRICH string solver [4]. The primary contribution of this paper is a complete and unified description of OSTRICH2's architecture, algorithms, and implementation as a modular and extensible system. Over the years, OSTRICH has undergone extensive internal development such as integrating new solving techniques, heuristics, and architectural changes, but many of these advances have not been previously documented or published. This includes, for instance, the use of word equation splitting strategies, character-count and length abstractions in preprocessing, and a portfolio-based orchestration of multiple solving engines. OSTRICH2 also includes a newly introduced solver based on algebraic data types (ADT-Str), described here for the first time. This paper is the first to present the overall system design of OSTRICH2, detailing how its components interact within a modular proof framework. We also provide the first full account of OSTRICH2's preprocessing strategies and rule infrastructure. These advances have radically enhanced the solver and resulted in significant performance improvements; notably, OSTRICH achieved first place in the QF_S track of the SMT-COMP 2023 competition. Together, this constitutes the first comprehensive system description of OSTRICH2. To this end, rather than being exhaustive, we aim for accessibility by taking the reader through illustrative examples (among others). In addition to unravelling the design of OSTRICH2, we also report our latest experimentation with the solver on SMT-LIB'25 benchmarks, showing its competitiveness, most notably on unsatisfiable instances.

The following is a list of main features of OSTRICH2:

(F1) SMT-LIB Unicode String theory inputs.
(F2) Native support of features of ECMAScript regular expressions (e.g., lookaround assertions, capture groups, and references; back-references are supported only in the replacement string of `replace` and `replace_all`, not in the matching expression).

https://doi.org/

(F3) Native support of `replace_all` and, more generally, complex string transductions using a more comprehensive *regular constraint propagation* strategy, which *combines* backward and forward regular propagations for a plethora of complex functions.

(F4) An extensive portfolio of solving strategies that includes a cost-enriched string solver (CE-Str) [24], a list-based ADT solver (ADT-Str), and preprocessings (length/character-count abstraction).

Related Tools.: Over the past decade, numerous string solvers have been proposed, employing techniques such as bit-vector encodings, automata-based propagation, and reductions to word equations. Several early or now-unsupported tools do not handle SMT-LIB 2.6 and are no longer actively maintained, including HAMPI [6], Kaluza [25], STRANGER [26], S3 [27], Norn [3], Trau [23], and members of the Z3Str family such as Z3str3 [28] and Z3str3RE [18]. These solvers contributed important techniques such as word-equation reasoning with advanced heuristics and direct regular expression support, but have not appeared in recent SMT-COMP competitions and are not included in our evaluation.

Our comparisons focus on state-of-the-art, actively maintained SMT solvers with SMT-LIB 2.6 support. Z3-alpha [13], which builds on Z3str4 [20] and Z3 [12], remains actively maintained and incorporates strategy synthesis over multiple string-solving backends. Z3 [12] and cvc5 [19] combine rewriting, word-equation decomposition, and regular constraint reasoning. Z3-Noodler [14] introduces a stabilization algorithm that propagated information on regular constraints to a selected word equation until the inferred regular languages on both sides stabilize. The solver is complete for the chain-free fragment but does not support string transductions or `replace_all`.

The original OSTRICH solver [4] implemented a backward regular constraint propagation strategy, providing completeness for the straight-line fragment and support for transductions. Subsequent extensions added the cost-enriched solver CE-Str [24] and ECMAScript-style regex support [29]. OSTRICH2, described in this paper, is the first to combine these capabilities with additional preprocessing, a newly implemented ADT-based solver, and a portfolio orchestration of multiple engines, as detailed in Section III.

Organization: Section II reviews the SMT-LIB Unicode Strings standard and our grammar; Section III reviews the architecture of OSTRICH2; Section IV details OSTRICH2's key algorithms; Section V discusses the completeness of the algorithms; Section VI shows that OSTRICH2 is extensible with user-defined string functions; Section VII reports evaluation results; and Section VIII concludes with directions for future work.

II. Specification Language

A. SMT-LIB Standard for Unicode Theory of Strings

We begin with the input language of OSTRICH2. It is based on SMT-LIB 2.6 [11] — in particular, including support for the SMT-LIB Unicode theory and Linear Integer Arithmetic constraints — but additionally also supports some "complex" string functions including transductions. We start by illustrating the supported language features by example and defer the formal description of the grammar to the end of the section.

1) SMT-LIB Constraints: Below is a minimal SMT-LIB script to show some core features. The script uses the quantifier-free string theory with linear integer arithmetic (`QF_SLIA`), contains three string variables `x`, `y`, and `z`, and one integer variable `l`. It `asserts` constraints on the variables that are explained below. The final statements check the satisfiability of the constraints and produce a satisfying assignment.

```
(set-option :produce-models true)
(set-logic QF_SLIA)

(declare-fun x () String)
(declare-fun y () String)
(declare-fun z () String)
(declare-fun k () Int)

(assert (= (str.len x) (+ k 1)))
(assert (= x (str.++ y z)))
(assert (str.in_re x (re.+ (re.union (str.to_re "a")
    ↪ (str.to_re "b")))))

(check-sat)
(get-model)
```

Constraints are written in a Lisp-style prefix notation. The first `assert` requires that the length of `x` is equal to `k+1`. The second requires that `x` is equal to the concatenation of `y` and `z`. The third requires that `x` belongs to the regular language $(\{\mathtt{a}\} \cup \{\mathtt{b}\})^+$, which represents non-empty sequences of `a` and `b` characters.

Standard Boolean connectives are supported. The example below requires `x` to either be the concatenation of `y` and `z` or the concatenation of `z` and `y`.

```
(assert (or (= x (str.++ y z))) (= x (str.++ z y))))
```

B. Extensions Beyond the SMT-LIB Standard

OSTRICH2 provides additional features to model string constraints in practical applications. A simple example is the `str.reverse` function. Below, we introduce the transducer, regular expression, and automata extensions.

1) Transducer-based Operations: OSTRICH2 supports transducers and prioritised finite-state transducers [30], [31], [29].

A transducer (a.k.a. a *rational transducer*) takes a single string as input and produces a single output string. They enable operations such as HTML encoding (e.g. replacing `&` with `&amp;`) to be defined. Transducers are written as recursive functions with two arguments: the input and the output. The example below shows a `toUpper` transduction. The base case (line 4) applies when both strings are empty. The recursive case (lines 5–11) asserts three conditions. First, both arguments are non-empty. Second, the head of the output (`y`) is equal to the upper case version of the head of the input (`x`). Third, the tails of the two arguments recursively satisfy `toUpper`. To obtain the upper case version of a character, arithmetic is performed on the character codes. In general, a transducer definition can contain multiple mutually recursive functions. The option `:parse-transducers` instructs OSTRICH2 to translate the recursive function definition to an internal transducer representation and apply an automata-based decision procedure to solve the constraints in lines 16–17.

```
1  (set-option :parse-transducers true)
2
3  (define-fun-rec toUpper ((x String) (y String)) Bool
4  (or (and (= x "") (= y ""))
5    (and (not (= x "")) (not (= y ""))
6         (= (char.code (str.head y))
7            (ite (and (<= 97 (char.code (str.head x)))
8                      (<= (char.code (str.head x)) 122))
9                 (- (char.code (str.head x)) 32)
10                (char.code (str.head x))))
11        (toUpper (str.tail x) (str.tail y)))))
12
13 (declare-fun x () String)
14 (declare-fun y () String)
15
16 (assert (= x "Hello World"))
17 (assert (toUpper x y))
```

Prioritised transducers additionally allow certain transitions to take precedence over other transitions. That is, a non-deterministic transduction may only succeed on one path if another cannot succeed. Such transducers are used internally in regular expression handling, described below, but can also be formulated using the `define-fun-rec` notation (not shown here). We discuss these below after we cover extended regular expressions.

2) Extended Regular Expression Support: The SMT-LIB standard defines a minimal set of regular expression constructs, all of which are supported by OSTRICH2. In addition, OSTRICH2 can also parse ECMAScript regular expressions, following the 2020 version [32], and supports regular expression features such as look-arounds that cannot be directly expressed in SMT-LIB. A simple usage of ECMAScript regular expressions is shown below, using the `re.from_ecma2020` function. The expression matches either sequences of characters or sequences of digits.

```
(assert (str.in_re w (re.from_ecma2020
    ↪ "[a-z]*|[0-9]*")))
```

To enable full support of ECMAScript character escaping, OSTRICH2 also supports single-quoted strings, in which the normal SMT-LIB character escaping is disabled and strings are passed unmodified to the regular expression parser:

```
(assert (str.in_re w (re.from_ecma2020 '\x24[0-9]+')))
```

One notable extension is the use of capture groups and references in replace and extract functions. The following example shows the replacement of any substring matching `src='.*'` with the text appearing between the single quotation marks. For example `<img src='image.png'>` would become `<img image.png>`. We use the `re.from_ecma2020` function for convenience; direct functions for regular expression operators—such as `re.capture`—are also available.

```
(assert (= x
          (str.replace_cg_all
             y
                (re.from_ecma2020 "src='(.*)'")
                (_ re.reference 1))))
```

There are several features to note about this example. The first argument of `str.replace_cg_all` is the string to be searched. The second is the regular expression that captures part of the string using the parenthesis notation `(...)`. The final argument is the *replacement pattern* that can be a concatenation of string literals and references to captured text. In this case, only the contents of the first (and only) capture group are used.

Importantly, when matching `src='(.*)'`, precedence rules must be respected. The `*` operator is *greedy* in ECMAScript (and many other regular expression languages), which means that it should match as many characters as possible before the remainder of the match begins. This can be a cause of errors when developers write regular expressions and a challenge for symbolic execution tools using a string constraint solver [33]. For example, if `y` contained the value `src='a' src='b'`, then there should only be one match of `src='(.*)'` rather than two. In the match, the captured value is `a' src='b`, that is, the longest string surrounded by `'` symbols. This semantics is respected by OSTRICH2 through the use of *prioritised* transducers [30], [31], [29]. In a prioritised transducer, steps of the transduction can be given priority over the alternatives. When matching against `src='a' src='b'` there is a choice whether the match of `(.*)` should stop at the second `'` or continue. Because `*` is greedy, the match of `(.*)` has priority and matching can only stop if continuing would fail. When replacing the pattern `src='(.*)'` with `src='(.*?)'`, using a *lazy* quantifier `*?`, instead the shortest string will be matched.

3) Automata Representations: OSTRICH2 also directly supports finite-state automata, which are often more convenient than regular expressions when integrating an SMT solver in applications. This automaton syntax is a bespoke extension specific to OSTRICH2. To the best of our knowledge, no other SMT solver supports this format. The function `re.from_automaton` parses a finite-state automaton given as a string. In the example below the automaton has

- an initial state `s0`,
- an accepting state `s1`,
- a transition `s0 -> s1 [0,100];` (accepting character codes 0–100),
- a loop `s1 -> s1 [0,65535];` (accepting any UTF-16 code unit).

Then `str.in_re` can be used to test whether the value of an expression belongs to the language of that automaton.

```
(assert (str.in_re x
       (re.from_automaton "automaton value_0 {init s0;
           ↪ s0 -> s1 [0, 100]; s1 ->s1[0,65535];
           ↪ accepting s1;};")))
```

A formal description is given in Appendix A.

4) SMT-LIB Standard for Unicode Theory of Strings: We begin by outlining the basic syntactic elements for our formulas and the key string and regular expression operations as defined in the SMT-LIB Unicode theory. The fragment below shows the core string and regular expression grammar supported natively by OSTRICH2. Functions not explicitly listed here are internally translated or reduced to equivalent formulas in this grammar. Operators that are extensions beyond SMT-LIB 2.6 are shown underlined.

Formulas:

$$\begin{aligned}
\phi &::= \neg\,\phi \mid \phi \wedge \phi \mid \phi \vee \phi \mid \mathit{Atom} \\
\mathit{Atom} &::= t_s \sim t_s \mid t_{ar} \sim t_{ar} \mid t_s \in t_{re} \mid \\
&\quad\ \mathit{StrPred} \mid \underline{\mathcal{T}(t_s, t_s)} \\
\mathit{StrPred} &::= \mathrm{prefixof}(t_s, t_s) \mid \mathrm{suffixof}(t_s, t_s) \mid \\
&\quad\ \mathrm{contains}(t_s, t_s) \\
t_s &::= \mathit{BaseStr} \mid \mathit{StrPos} \mid \mathit{StrRep} \mid \\
&\quad\ \underline{\mathrm{reverse}(t_s)} \\
\mathit{BaseStr} &::= c_{str} \mid x_{str} \mid \mathrm{concat}(t_s, t_s) \\
\mathit{StrPos} &::= \mathrm{at}(t_s, t_{ar}) \mid \mathrm{substr}(t_s, t_{ar}, t_{ar}) \\
\mathit{StrRep} &::= \mathrm{rep}(t_s, t_s, t_s) \mid \mathrm{rep\_all}(t_s, t_s, t_s) \mid \\
&\quad\ \mathrm{rep\_re}(t_s, t_{re}, t_s) \mid \mathrm{rep\_re\_all}(t_s, t_{re}, t_s) \\
t_{ar} &::= c_{int} \mid x_{int} \mid t_{ar} + t_{ar} \mid |t_s| \mid \\
&\quad\ \mathrm{indexof}(t_s, t_s, t_{ar}) \\
t_{re} &::= \emptyset \mid \Sigma \mid \Sigma^* \mid \mathrm{toRE}(t_s) \mid t_{re} \cdot t_{re} \mid \\
&\quad\ t_{re} \cup t_{re} \mid t_{re} \cap t_{re} \mid t_{re}^*
\end{aligned}$$

Explanation

- **Formulas (ϕ) and Atoms:** The Boolean formulas ϕ are constructed using the standard logical connectives (negation $\neg$, conjunction $\wedge$, and disjunction $\vee$) along with atomic formulas. The atomic formulas include comparisons on string terms t_s (via a placeholder relation $\sim$), arithmetic terms t_{ar} (also using $\sim$), the membership test $t_s \in t_{re}$, and the dedicated string predicates defined by $\mathit{StrPred}$. Finally, $\underline{\mathcal{T}(t_s, t_s)}$ encodes a transducer. The complete grammar for transducers is provided in Appendix B.
- **String Predicates ($\mathit{StrPred}$):** This subclass of atoms is dedicated to predicates that operate specifically on strings. It includes:
  - $\mathrm{prefixof}(t_s, t_s)$: Checks if the first string is a prefix of the second.
  - $\mathrm{suffixof}(t_s, t_s)$: Checks if the first string is a suffix of the second.
  - $\mathrm{contains}(t_s, t_s)$: Checks if the first string is contained within the second.
- **String Terms (t_s):** The nonterminal t_s is divided into three subclasses and one string function:
  - $\mathit{BaseStr}$**:** Represents the basic string values. This includes string constants c_s, string variables x_s, or the concatenation of two string terms using the operator concat.
  - StrPos**:** Represents string functions that require an integer parameter. These include:
    - $\mathrm{at}(t_s, t_{ar})$: Returns the character (as a string) at a specified position.
    - $\mathrm{substr}(t_s, t_{ar}, t_{ar})$: Returns a substring starting at a given position with a specified length.
  - StrRep**:** Contains the string replacement functions, abbreviated here for conciseness. They include:
    - $\mathrm{rep}(t_s, t_s, t_s)$: Literal replacement.
    - $\mathrm{rep\_all}(t_s, t_s, t_s)$: Replace all occurrences (literal).
    - $\mathrm{rep\_re}(t_s, t_{re}, t_s)$: Regex-based replacement.
    - $\mathrm{rep\_re\_all}(t_s, t_{re}, t_s)$: Replace all occurrences based on a regex.

  Finally, the operator reverse reverses a string.
- **Arithmetic Terms (t_{ar}):** The arithmetic expressions include integer constants c_{int}, integer variables x_{int}, the sum of two arithmetic expressions $t_{ar} + t_{ar}$, the length function $|t_s|$ (which returns the length of a string), and the function $\mathrm{indexof}(t_s, t_s, t_{ar})$, which returns the index of one string within another as an integer.
- **Regular Expressions (t_{re}):** Regular expressions are defined in a manner consistent with standard mathematical notation:
  - $\emptyset$ denotes the empty language.
  - Σ is the alphabet, and Σ^* represents its Kleene closure.
  - $\mathrm{toRE}(t_s)$ converts a string term into its corresponding regular expression.
  - The operators $\cdot$ (concatenation), $\cup$ (union), $\cap$ (intersection), and * (Kleene star) are used to build more complex regular expressions.

III. System Architecture

The overall architecture of OSTRICH2 is depicted schematically in Figure 1. OSTRICH2 runs the three solvers (ADT-Str, RCP, and CE-Str) sequentially in a time-sliced configuration. Each solver receives the same input problem and runs independently for a fixed share of the total timeout (e.g. 20 seconds each for a 60-second limit). To reduce repeated work, OSTRICH2 performs common subexpression sharing for selected functions while representing all other expressions separately in tree form. Once started, no further information is shared between solvers as there is currently no established methodology in the literature for aggressive or global sharing of string expressions.

At its core, OSTRICH2 builds on the SMT solver Princess [34], which provides the logical reasoning framework and support for theories such as linear integer arithmetic. OSTRICH2 provides pre-processing and support for the quantifier-free string theories `QF_S` (pure strings) and `QF_SLIA` (strings with linear integer arithmetic). Princess also provides support for algebraic data-types (ADTs), on top of which the ADT-Str string solver is implemented.

The second and third solver are the Regular Constraint Propagation (RCP) and Cost-Enriched String (CE-Str) engine, respectively. These share similar architectures. They begin with a shared string preprocessor, which processes the SMT-LIB input and forwards it to the SMT Core of Princess. Details of the string preprocessing are given in the next section. The SMT Core coordinates with the LIA Solver for linear integer arithmetic and with a solver engine for string-specific reasoning.

Both solvers uses automata- and tranducer-based representations of sets of strings and supported string functions. A main loop applies proof rules like Automata Intersection, Forward (FWD) and Backward (BWD) Propagation, Nielsen Splitting, and other string-related inference rules. Details of the rules are given in the next section. A string database and an automata database are used for efficiently storing and retrieving string and automata data. The automata database is based on the BRICS Automata Library [35], which provides efficient automaton operations.

For the purposes of presentation, we abstract the constraints received from the SMT core into the following *normal form*:

$$\begin{aligned} S &::= A \mid \neg A \mid S \wedge S \\ A &::= f(x_1, \ldots, x_n) \mid x = g(x_1, \ldots, x_n) \mid \\ &\quad\;\; x \in L \mid x = \sum_i d_i x_i \ . \end{aligned}$$

In this notation, $x, x_1, \ldots, x_n$ are variables of type string or integer, and each d_i is an integer constant. The symbol f denotes a string predicate such as $\mathsf{prefixof}(x_1, x_2)$ or $\mathsf{suffixof}(x_1, x_2)$. The symbol g denotes a string function that takes $x_1, \ldots, x_n$ as inputs and produces the string x as output, for example $x = \mathsf{concat}(x_1, x_2)$. The notation $x \in L$ expresses that the value of x belongs to a regular language L, which is typically given by a regular expression in SMT-LIB syntax. Finally, the form $x = \sum_i d_i x_i$ represents a linear integer constraint, most commonly arising from length constraints, where the x_i are integer variables and the d_i are constant coefficients. We sometimes write $|t|$ to denote the length of the word represented by the term t. For example, the constraint

```
(assert (and (= (str.++ x y) (str.++ y x))
             (str.in_re x (re.from_ecma2020 "a*ba*"))
             (str.in_re y (re.from_ecma2020 "a*ca*"))))
```

has the following normal form using an additional variable z

$$z = \mathrm{concat}(x, y) \wedge z = \mathrm{concat}(y, x) \wedge x \in a^*ba^* \wedge y \in a^*ca^* \ .$$

In the sections below, we give an overview of the three main solver engines. These are: ADT-Str (algebraic datatypes), RCP (regular constraint propagation), and CE-Str (cost-enriched strings).

A. *ADT-Str: List-Based Solver*

The ADT-Str solver builds on the decision procedure for algebraic data-types (ADTs) with size catamorphism implemented in Princess [36]. Algebraic data-types are used to represent strings using the standard encoding of lists with `nil` and `cons` constructors. The length of a string is computed using the built-in `size` function provided by the ADT solver, mapping every constructor term to the number of constructor occurrences. Other SMT-LIB functions on strings, for instance substring, concatenation, etc., are in ADT-Str encoded using uninterpreted functions and axioms capturing the recursive definition of the string functions. Regular expression matching is implemented using Brzozowski derivatives [37].

ADT-Str is useful, in particular, for computing solutions of string constraints that are outside of the fragments for which the other solvers are complete. ADT-Str can easily handle certain functions that are hard for our propagation algorithms. Those functions include, among others, string-to-integer conversion, and functions like substring and indexof that calculate with integer offsets.

B. *RCP: Regular Constraint Propagation*

Regular Constraint Propagation (RCP) is the newest algorithm that has been implemented in OSTRICH2, based on a subset of proof rules in our paper [29]. The main goal of RCP is to *prove unsatisfiability of the input constraint*. The algorithm handles string functions like concatenation, replace, replaceall, and regular constraints. Other string functions (e.g. reverse, one-way and two-way transducers) are also permitted. These functions permit either exact pre-image or exact post-image computation of regular constraints or both. In general, these images need not be exact, but must at least overapproximate the true pre/post image.

The main idea behind RCP is to propagate regular constraints of the form $x \in L$ to other string variables through forward and backward propagations using the RCP inference rule described in the next section.

ECMAScript Regular Expressions: All three solvers in OSTRICH2 support the SMT-LIB regular expression operators. Full support for ECMAScript regular expression features, including look-arounds, capture groups, and greedy/lazy quantifiers, is implemented in the RCP solver, following the approach in earlier work [29], [38]. Back-references are supported only in the replacement string of `replace` and `replace_all`, not in the matching expression. This implementation uses prioritised transducers to preserve ECMAScript semantics and an intermediate translation to alternating two-way automata for look-arounds. We refer to [29], [38] for full technical details.

C. *CE-Str: Cost-Enriched String Engine*

The main algorithm of CE-Str (based on [24]) applies backward regular constraint propagation. It uses cost-enriched finite automata (CEFAs) instead of standard finite automata to represent sets of words. These automata are able to capture numerical information about accepted strings, such as the string length, the number of characters appearing before a transition is fired, and so on. This allows length constraints on strings and functions like indexof to be directly supported.

In addition, CE-Str can handle counting operators more efficiently by leveraging the numerical information encoded in CEFAs. For instance, the regex $a^{\{1,1000\}}$, which accepts strings consisting of 1 to 1000 repetitions of the character a, can be represented as a CEFA with just one state and one

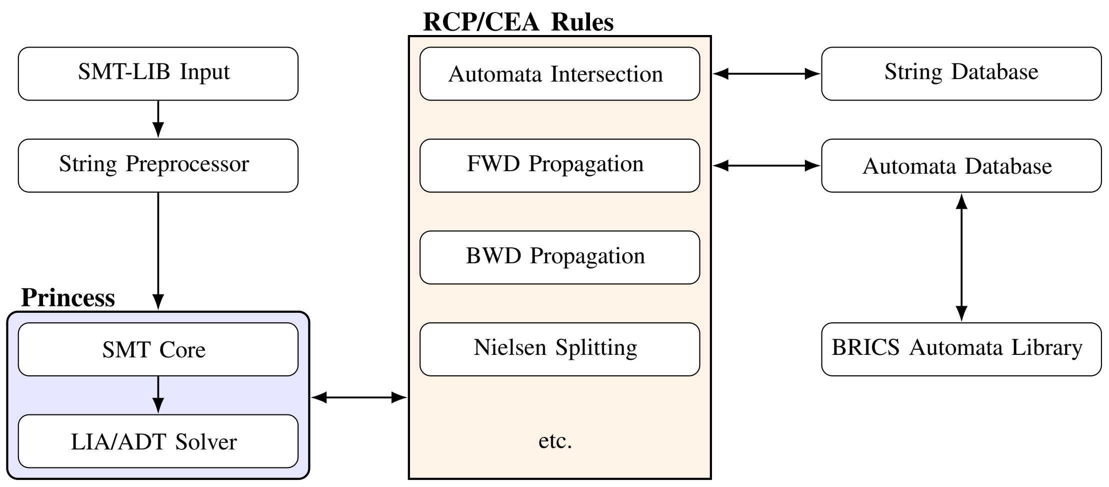

Figure 1. Overall architecture of OSTRICH2: the SMT-LIB input is handled by our string preprocessor and Princess SMT core (with LIA and ADT-Str), while the RCP and CE-Str solvers consist of inference rules that are repeatedly applied using string and automata databases.

transition—compared to the 1000 states and 1000 transitions required by a standard finite automaton.

CE-Str supports a wide range of string functions, including concatenation, replace, replaceall, substring, indexof and length. It provides completeness guarantees for the straight-line fragment of these functions. CE-Str complements RCP and ADT-Str. It performs well on string constraints involving integer type but struggles when the constraints are outside its decidable fragment.

IV. String Theory Algorithms

In this section we describe the core inference rules and algorithmic components that OSTRICH2 applies to solve string constraints. We first present the preprocessing steps shared by the RCP and CE-Str solvers. We then present the *inference rules* applied in the main loops of RCP and CE-Str.

A. Preprocessing

Before applying the core inference rules, OSTRICH2 performs a series of preprocessing steps to enrich the constraint set with auxiliary information and simplify trivial cases. The simplification rules are given below. First we describe the simplification rules used by both RCP and CE-Str, then we describe the additional rules used by RCP only.

1) Common Preprocessing:

Prefix/Contains/Suffix Simplification: We apply the following rewrites when one argument is concrete:

- $\mathrm{prefixof}(s,t)$ / $\mathrm{suffixof}(s,t)$ with concrete s: replace by
$$t \in L(\mathtt{s.\star}) \quad / \quad t \in L(\mathtt{.\star s}).$$
- $\mathrm{contains}(t,s)$ with concrete s: replace by
$$t \in L(\mathtt{.\star s.\star}).$$

In *positive* contexts we instead encode:

$$\begin{array}{rcl}\mathrm{prefixof}(s,t) & \rightsquigarrow & t = \mathrm{concat}(s,u)\\ \mathrm{suffixof}(s,t) & \rightsquigarrow & t = \mathrm{concat}(u,s)\end{array}$$

for fresh u. Finally, $\mathrm{prefixof}(t,t)$, $\mathrm{contains}(t,t)$, $\mathrm{suffixof}(t,t)$ are *trivially true* and replaced by the Boolean constant true.

Common Sub-Expression Elimination: While OSTRICH2 does not apply aggressive sharing of terms and expressions, it is useful to merge repeated occurrences of certain operators that are relatively expensive to handle, including expressions $\mathrm{indexof}(x,y,i)$. For expressions e of this kind that occur multiple times in a formula, OSTRICH2 introduces a fresh variable k and adds the equation $k = e$. All occurrences of e in the formula are then rewritten as k. This rewriting ensures that the solver only sees one copy of each complex sub-expression and speeds up subsequent reasoning.

2) RCP Preprocessing: The next simplifications are currently used by RCP only.

a) Length and Character-Count Approximation: Infers approximate bounds on string lengths and character counts for complex string functions.

- *Derived length constraints:* derive length constraints and character count approximations from word equations and other string functions. E.g. $|x| = |y| + |z|$ is added for $x = \mathrm{concat}(y,z)$.
- *Automaton-transition analysis:* inspect each transition in an automaton A and, for any character a not appearing on any transition, add the char count constraint $|x_a| = 0$ for every x constrained by A.
- *Index-of range constraints:* for each occurrence of $\mathrm{indexof}(x,y,i)$ (which returns the first index of y in x after position i), add implied constraints on the result. E.g., $-1 \le \mathrm{indexof}(x,y,i) \le |x|$ captures that the result must be either be -1 or in the interval $[0,|x|]$.

b) Index-of/Substr/At Rewriting: Translate substr, at, and indexof into equivalent combinations of string concatenation, length constraints, and regex-membership tests by introducing fresh variables and constraints to encode the positional semantics.

We illustrate one such rewrite for $r = \mathrm{substr}(s, i, n)$, which asserts that r contains the longest contiguous substring of s of length at most n starting at position i. It can be split into three cases.

First we introduce fresh string variables p, r, q with

$$s = \mathrm{concat}(p, r, q) \wedge |p| = i,$$

and then distinguish:

- If $i < 0$, $n \leq 0$, or $i \geq |s|$, then $r = \varepsilon$.
- If $0 \leq i < |s|$ and $i + n \leq |s|$, then $|r| = n$, so r is exactly the length-n slice of s starting at position i.
- If $0 \leq i < |s|$ but $i + n > |s|$, then $|r| = |s| - i$, so r is the substring from position i to the end of s.

B. Inprocessing Rules

There is also a set of lightweight simplification rules that are applied to expressions during proving. Such rules are applied in the local context of a proof goal and are often able to significantly simplify expressions, for instance by evaluating function applications with known arguments or discovering obvious contractions. During inprocessing, equations and assignments of values to variables are propagated to other constraints. There are also certain preprocessing rules that are applied again during inprocessing; for instance, $\neg\ \mathrm{suffixof}(s, t)$ can be turned into a regular expression constraint as soon as one of the arguments s, t is a known string, but has to be kept unchanged before that.

C. Inference Rules

The main loops of the RCP and CE-Str repeatedly apply proof rules until a branch is closed or a model is found. We discuss the principal rules in this section. The additional rules omitted here that which cover both string-specific reasoning and SMT-core inferences are described in previous work [29], and together with the rules below constitute a complete proof system.

The RCP algorithm assigns priorities to each possible rule application based on an estimation of the workload (e.g. the size of the involved automata) and selects the rule application with the highest priority first. Fairness is ensured by penalizing newer rule applications, and therefore preferring rule applications that have resided in the waiting queue the longest. Priorities are computed as the weighted sum of several criteria:

- *Concrete-argument:* any rule whose input or output language is a ground string is given high priority.
- *Information-gain penalty (backward only):* rules whose input automata are universal (i.e. accept all words, e.g. $x \in \Sigma^*$) yield little new information in backward propagation and given low priority.
- *Exactness adjustment (forward only):* forward rules for functions without an exact post-image (e.g. replaceall with symbolic patterns) are given low priority.
- *Cost-based penalty:* a weight proportional to the combined size of the input and result automata is subtracted.

Differences between the two solvers in their application of the rules is included in the descriptions. The proof rules may cause branching in the proof search. If a satisfying assignment is found on a branch, it witnesses that the constraint is satisfiable. The Close rule detects unsatisfiable constraints. If all branches are closed, the constraint is unsatisfiable.

a) Regex-to-Automata: OSTRICH2 constructs an automaton A for each regular membership constraint in $x \in L$. The automaton is stored in the automaton database, which detects equivalent automata to avoid duplication. In the remaining rules, we make the automaton explicit by writing $x \in L(A)$. The CE-Str solver creates cost-enriched automata. Initially these automata do not track any costs. Cost tracking is introduced during the RCP rule below.

b) BreakCyclicEquations: OSTRICH2 detects strongly connected components in the variable-dependency graph induced by concatenation equations of the form

$$x = \mathrm{concat}(y, z) \quad \text{and} \quad y = \mathrm{concat}(a, x).$$

For each cycle it removes one equation and, for every remaining equation $v = \mathrm{concat}(u, w)$ in that cycle, adds a trivial emptiness constraint on one argument (e.g. $w = \varepsilon$ or $u = \varepsilon$) to break the dependency.

For example, from

$$x = \mathrm{concat}(y, z) \quad \wedge \quad y = \mathrm{concat}(a, x)$$

we might remove the first equation and introduce

$$y = \mathrm{concat}(a, x), \quad z = \varepsilon, \quad a = \varepsilon,$$

yielding an acyclic set of equations. This transformation is *sound*, because in any finite model of the original cyclic equations at least one concatenation argument must be empty, and *complete*, since any solution of the resulting acyclic system (with the added emptiness constraints) automatically satisfies the dropped equations.

c) Equation Decomposition: OSTRICH2 uses two simple heuristics to simplify word equations. In what follows, z, x_1, x_2, y_1, y_2 are arbitrary string terms (each may be a variable or a constant). First, when one side is a constant string w and the other is a concatenation

$$w = \mathrm{concat}(x_1, x_2),$$

in which the length of x_1 is known, OSTRICH2 matches the first $|x_1|$ characters of w with x_1 and the remaining characters with x_2. Second, if the same variable appears in two concatenations,

$$z = \mathrm{concat}(x_1, x_2) \quad \text{and} \quad z = \mathrm{concat}(y_1, y_2),$$

then whenever $|x_1| = |y_1|$ we infer $x_1 = y_1$ and $x_2 = y_2$. These heuristics often resolve equations without full case splitting.

d) Close: If for some variable x we have constraints

$$x \in L(A_1) \wedge \cdots \wedge x \in L(A_k)$$

and $\bigcap_{i=1}^{k} L(A_i) = \emptyset$, OSTRICH2 derives a contradiction and close the branch.

e) Intersection: When the `+eager` flag is on, OSTRICH2 maintains at most one automaton per variable by replacing $x \in L(A) \wedge x \in L(B)$ with $x \in L(A \cap B)$.

f) LengthAbstraction: From length inequalities (e.g. $|x| \leq |y| + 5$), derive lower/upper bounds on $|x|$, then assert $x \in L(A)$ where A accepts any word within the derived length bounds.

g) RCP (Regular Constraint Propagation): Propagate $x_1 \in L_1, \ldots, x_n \in L_n$ forwards through string functions $x = f(x_1, \ldots, x_n)$, or $x \in L$ backwards through f.

For example, on $z = concat(x, y) \wedge z = concat(y, x) \wedge x \in a^*ba^* \wedge y \in a^*ca^*$ we can propagate $x \in a^*ba^*$ and $y \in a^*ca^*$ forwards (from input to output) through the string concatenation function $concat$ in $z = concat(x, y)$ to obtain $z \in a^*ba^*a^*ca^*$. Similarly, we can propagate forwards through $z = concat(y, x)$ and derive $z \in a^*ca^*a^*ba^*$. Since there are no words matching both $a^*ba^*a^*ca^*$ and $a^*ca^*a^*ba^*$ we can conclude—using the Close rule—that the constraint is unsatisfiable.

Backwards propagation may result in branching. For example, if $x = \mathrm{concat}(y, z)$ and $x = ab$, then either $y = ab \wedge z = \varepsilon$ or $y = a \wedge z = b$ or $y = \varepsilon \wedge z = ab$.

When the variable constraints are given by standard finite automata, a rich selection of string functions support exact forwards and backwards propagation. These include concat, $\mathrm{reverse}$, and $\mathrm{replace}$ and $\mathrm{replaceall}$, where the replacement pattern can include string variables or references to capture groups in the search pattern. In general, any function can be supported using over-approximations of the pre- and post-images, but without completeness guarantees.

The CE-Str solver only applies backwards propagation and represents pre-images using cost-enriched automata. This allows functions such as $\mathrm{indexof}$ and length to be supported (by introducing costs to the automata), but restricts, for example, which versions of $\mathrm{replace}$ and $\mathrm{replaceall}$ can be analysed precisely.

h) NielsenSplitter: OSTRICH2 invokes Nielsen's transformation [39], [40] on any pair of equations in our normal form

$$z = \mathrm{concat}(x_1, x_2) \quad \text{and} \quad z = \mathrm{concat}(y_1, y_2),$$

where each of z, x_i, y_i may be a variable or a constant. We introduce a fresh string variable w and split the proof into two *prefix-alignment* cases, guided by current length information in a similar way as was done in Norn [3]. That is

$$|x_1| \geq |y_1|, \quad x_1 = \mathrm{concat}(y_1,\, w), \quad \mathrm{concat}(w,\, x_2) = y_2$$

or

$$|y_1| \geq |x_1|, \quad y_1 = \mathrm{concat}(x_1,\, w), \quad \mathrm{concat}(w,\, y_2) = x_2.$$

In each branch we align the longer prefix against the shorter one, adding both the corresponding concatenation equalities and the derived length equality (e.g. $|x_1| = |y_1| + |w|$ in the first case).

i) String-Integer-Conversions: For handling expressions $\mathrm{str\_to\_int}(s) = n$ or $\mathrm{int\_to\_str}(n) = s$, OSTRICH2 systematically explores the possible values of n. As soon as n has a concrete value, inprocessing rules (Section IV-B) apply that replace the function application with an equation or a regular expression constraint describing the possible values of s. To make this exploration perform well in practical cases, the LIA solver utilized in OSTRICH2 applies interval constraint propagation to narrow down the range of possible values of n as much as possible [41]. The interval for n is then subdivided, until eventually only one possible value for n remains. This search will not terminate in general, since the domain of n can be infinite, but it tends to derive solutions or contradictions quickly in many practical cases.

j) Index Computation: When handling $\mathrm{indexof}(s, p, k)$ on two concrete strings s, p, OSTRICH2 systematically explores possible values of k, taking into account the fact that any value k not satisfying $0 \leq k \leq |s| - |p|$ will necessarily lead to the result -1. In the "in-range" branch OSTRICH2 applies interval constraint propagation to determine the possible values of k, sub-dividing the interval of values until a concrete value k has been chosen. As soon as all arguments of $\mathrm{indexof}$ have concrete values, inprocessing rules (Section IV-B) can evaluate the function.

k) Cut Rule: When all other rules have been exhausted, OSTRICH2 applies cuts to introduce a candidate solution for a string variable x. For this, OSTRICH2 collects every regular-language membership constraint $x \in L_i$ together with any length bounds $\ell \leq |x| \leq u$. From the intersection of the L_i (and respecting ℓ, u), OSTRICH2 extracts a single accepted word w via a standard automaton search. We then split on the two exhaustive cases

$$x = w \quad \text{vs.} \quad x \notin \{w\}.$$

This rule is always sound because any satisfying assignment for x either equals the chosen w or lies outside $\{w\}$ and guarantees that each string variable will eventually be grounded to a concrete value (or shown impossible).

V. Completeness Results

The RCP solver is (in principle) complete for the straight-line fragment [4] and chain-free fragment [42] of string constraints, in the sense that the underlying proof system can show unsatisfiability of unsatisfiable instances from these fragments, when applying the rules in an appropriate (easy) order. However, the usage of priorities means the rules may be applied in a different order, causing the proof to fail. In practice, failures for these fragments are rare.

In the straight-line fragment we require that the constraints are in normal form as in Section IV-C, and variables can be ordered $x_1, \ldots, x_n$ such that for x_i there is exactly one equational constraint $x_i = f_i(x_1, \ldots, x_{i-1})$, where f_i is a string function such that the pre-image of a regular set is regular. The definition of the chain-free fragment is a bit involved, in essence it relaxes the assumption on the form of the equation and the number of equations, but still requires that there are no "cycles" of dependencies. For the straight-line fragment, we can propagate backwards and the instance is unsatisfiable if and only if all branches are closed using Close rule. For the chain-free fragment [42] we can adapt the original proof (which works for a different solver) to the rules used in OSTRICH2.

The CE-Str solver also provides a completeness guarantee for a version of the straight-line fragment that includes string constraints and linear integer arithmetic [24]. The supported string functions are those where the pre-image of cost-enriched automata constraints can be expressed using cost-enriched automata. This includes indexof and length as well as some versions of replace and replaceall.

VI. Extensibility

An important feature of OSTRICH2 is the possibility to extend the solver with minimal effort. OSTRICH2's `PreOp` trait lets users add new string functions by overriding three core methods. This can be done in either Scala (the main language used to implement OSTRICH2) or in Java. Below we show each method in turn, together with the `ReversePreOp` implementation[1].

First, the method `apply` computes the *pre-image* of a regular language under f, which is the core operation needed for backward propagation in RCP. The method is passed any existing automata constraints on the r arguments (which may be ignored) and the automaton $\mathcal{A}$ representing the result language. The method returns an iterator over tuples of automata whose images under f lie inside $\mathcal{A}$, plus (optionally) the subset of the input constraints actually used.

```scala
override def apply(
  argCs: Seq[Seq[Automaton]],
  resultC: Automaton
) : (Iterator[Seq[Automaton]], Seq[Seq[Automaton]]) = {
  (Iterator(Seq(ReverseAutomaton(resultC))), List())
}
```

Second, the method `eval` performs *concrete* evaluation on ground strings: when all arguments are known, it simply returns the result of applying f.

```scala
override def eval(
  args: Seq[Seq[Int]]
): Option[Seq[Int]] = Some(args.head.reverse)
```

Third, `forwardApprox` computes a (sound) *post-image* of input languages under f, used for forward propagation. While returning the universal automaton is always correct, a tighter approximation greatly improves performance. For `str.reverse`, one can exactly compute this by intersecting any input automata and then reversing the resulting automaton.

```scala
override def forwardApprox(
  argCs: Seq[Seq[Automaton]]
): Automaton = {
  val prod = ProductAutomaton(argCs)
  ReverseAutomaton(prod)
}
```

[1]This snippet is abbreviated for clarity. The full implementation in `ReversePreOp.scala` includes additional automaton-type matching and error handling.

Finally, the new string function has to be registered in the core string theory class of OSTRICH2. In the file `OstrichStringTheory.scala`, we register `str.reverse` by first declaring its SMT-side function symbol:

```scala
val str_reverse = MonoSortedIFunction(
  "str.reverse",
  List(StringSort), StringSort, true, false
)
```

We then add it to the `extraStringFunctions` list, pairing the name, the `IFunction`, our `ReversePreOp` implementation, and the simple lambdas that pick out the input and output terms:

```scala
val extraStringFunctions = List(
  ("str.reverse", str_reverse, ReversePreOp,a =>
     List(a(0)),a => a(1)))
```

With these two lines in place, any occurrence of `str.reverse` in the SMT-LIB script is recognized by OSTRICH2 and automatically dispatched to our `ReversePreOp` for pre-image computation, concrete evaluation, and forward-approximation.

VII. Experiments

A. Benchmark suites and experimental setup

We evaluate our solver on a representative selection of SMT-LIB benchmarks, including all instances from the SMT-LIB 2025 benchmark release[2]. From the combined pool of roughly 100 000 `QF_S` and `QF_SLIA` problems, we randomly sample 2 000 instances proportional to each track's share of the total, so that our test set reflects the underlying distribution of problem types. 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.

ECMAScript Regex Benchmarks.: Our experimental evaluation in this paper does not include benchmarks making extensive use of ECMAScript-specific regular expression features such as look-arounds or capture groups. A large-scale evaluation of this functionality was presented in [38], using a benchmark set of approximately 8,800 ECMAScript-style regex patterns extracted from real-world web forms. The correctness of the semantics was validated in [29] by comparing OSTRICH's results against JavaScript's reference implementation. The ECMAScript regex component in OSTRICH2 is identical to that used in those earlier evaluations. Therefore, we omit repeating those experiments here and focus on broader SMT-LIB benchmarks.

B. Performance evaluation

The experiments are conducted on the following solvers: cvc5 1.2.0 [19], Z3 4.15 [12], Z3-alpha [13], Z3-Noodler 1.3.0 [14], OSTRICH1 (the original OSTRICH) [4], and different engines of OSTRICH2[3] and OSTRICH2 running

[2]https://zenodo.org/communities/smt-lib/
[3]https://doi.org/10.5281/zenodo.15378521

the engines in a time-sliced portfolio using the flag. For the RCP engine we evaluate two complementary configurations: one mode that combines forward and backward propagation without Nielsen splitting (using the flags `+F+B-N`), and another mode (using the flags `-F+B+N`) that disables forward propagation in order to emphasize Nielsen's equation decomposition. In practice, combining forward propagation and splitting yields limited synergy: the extra word equations generated by Nielsen splitting create significant propagation overhead, so omitting forward propagation often improves overall solving performance.

	cvc5	Z3	Z3-alpha	Z3-Noodler	OSTRICH1
SAT	1155	1097	1123	**1201**	825
UNSAT	729	728	728	**744**	664
unknown/timeout	116	175	149	**55**	511
Solved	1884	1825	1851	**1945**	1489
Total time [s]	7770	12158	10523	**3660**	35711

Figure 2. The experiment result of cvc5, Z3, Z3-alpha, Z3-Noodler and OSTRICH1 on the SMT-LIB'25 benchmarks. Total time (in seconds) includes all instances, with each unknown/timeout counted as 60s.

	OSTRICH2	ADT	CE	RCP-F+B+N	RCP+F+B-N
SAT	**1147**	610	970	1067	1042
UNSAT	**758**	619	679	747	750
unknown/timeout	**95**	771	351	186	208
Solved	**1905**	1229	1649	1814	1792
Total time [s]	**14824**	53689	24600	17680	19900

Figure 3. The experiment result of different engines of OSTRICH2 on the SMT-LIB'25 benchmarks.The flags `+F+B-N` enable forwards/backwards propagation and disable the NielsenSplitter. The flags `-F+B+N` disable forwards propagation and enable backwards propagation and the NielsenSplitter. Total time (in seconds) includes all instances, with each unknown/timeout counted as 60s.

Figure 2 shows that among the off-the-shelf solvers, Z3-Noodler leads with 1 945 solved problems, followed by cvc5 (1 884), Z3-alpha (1 851), Z3 (1 825), and OSTRICH1 (1 489). Figure 3 shows that both RCP configurations (1 814 and 1 792 solves) and CE-Str (1 649) represent clear upgrades over OSTRICH1, whereas the ADT-Str engine (1 229) trails—largely due to its simple algorithm. These per-engine gains can be attributed to improved preprocessing, an extended set of proof rules, and, for CE-Str, a different underlying solving technique.

In the full OSTRICH2 configuration, these upgraded engines are run in a time-sliced portfolio, which not only inherits the improvements of the individual modes but also exploits their complementary strengths, as each engine solves benchmarks that the others miss. This complementarity lifts OSTRICH2's overall performance close to the top solvers in the field.
When we combine all four engines in a time-sliced portfolio (15 seconds for each mode for a 60-second limit), OSTRICH2 solves 1 905 problems, putting it just behind Z3-Noodler and ahead of the rest of the field. The portfolio not only inherits the individual gains of each engine but also benefits from their complementary strengths, as each solves benchmarks the others miss. We further observe that Z3-Noodler excels on the SAT instances, while OSTRICH2 performs particularly well on the UNSAT cases.

Finally, although Z3-Noodler does not support `replace_all`, only 2.4% of SMT-LIB instances use this operator, so its absence has only a modest effect on overall rankings. Figure 4 shows the results on the benchmark set restricted to instances without `replace_all`. Among the 35 `replace_all` benchmarks in our SMT-LIB sample, OSTRICH2 solves 25, while cvc5, Z3, and Z3-alpha each solve only two instances, and Z3-Noodler solves none. This shows that most solvers provide limited support for this operator. When we exclude the `replace_all` benchmarks, the rankings remain broadly unchanged; OSTRICH2 drops slightly below cvc5 in relative performance.

We remark that, since our evaluation uses a stratified random sample of 2 000 instances drawn from the full pool of roughly 100 000 `QF_S` and `QF_SLIA` benchmarks, there is inevitably some statistical variance in the exact per-solver ranking; nonetheless, the fact that OSTRICH2 solves almost as many problems as the leader demonstrates its overall competitiveness.

	cvc5	Z3	Z3-alpha	Z3-Noodler	OSTRICH2
SAT	1155	1097	1123	**1201**	1141
UNSAT	727	726	726	**744**	739
unknown/timeout	83	142	116	**20**	85
Solved	1882	1823	1849	**1945**	1880
Total time [s]	5790	10174	8536	**1617**	14070

Figure 4. The experiment result of cvc5, Z3, Z3-alpha, Z3-Noodler and OSTRICH2 on the SMT-LIB'25 benchmarks **excluding** `replace_all`. Total time (in seconds) includes all instances, with each unknown/timeout counted as 60s.

VIII. Conclusion

This paper has strived to provide a concise introduction of the string solver OSTRICH2, which inputs constraints in an extension of the SMT-LIB theory of Unicode Strings with transducers (through mutual recursion) and ECMAScript regular expressions, among others. We have also empirically demonstrated its competitiveness with other string solvers on SMT-COMP benchmarks, particularly on unsatisfiable instances. Interested readers, who are interested in participating in the development and/or applications of OSTRICH2, are wholeheartedly encouraged to get in touch with us.

Acknowledgements

We thank EPSRC [EP/T00021X/1], European Research Council (LASD, 101089343), and the Swedish Research Council (grant 2021-06327) for their support.

References

[1] C. Kern, "Securing the tangled web," *Commun. ACM*, vol. 57, no. 9, pp. 38–47, Sep. 2014.

[2] A. W. Lin and P. Barceló, "String solving with word equations and transducers: towards a logic for analysing mutation XSS," in *POPL*. ACM, 2016, pp. 123–136.

[3] P. A. Abdulla, M. F. Atig, Y. Chen, L. Holík, A. Rezine, P. Rümmer, and J. Stenman, "Norn: An SMT solver for string constraints," in *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part I*, ser. Lecture Notes in Computer Science, D. Kroening and C. S. Pasareanu, Eds., vol. 9206. Springer, 2015, pp. 462–469. [Online]. Available: https://doi.org/10.1007/978-3-319-21690-4_29

[4] T. Chen, M. Hague, A. W. Lin, P. Rümmer, and Z. Wu, "Decision procedures for path feasibility of string-manipulating programs with complex operations," *Proceedings of the ACM on Programming Languages*, vol. 3, no. POPL, pp. 1–30, 2019.

[5] P. Saxena, D. Akhawe, S. Hanna, F. Mao, S. McCamant, and D. Song, "A symbolic execution framework for javascript," in *S&P*, 2010, pp. 513–528.

[6] A. Kiezun, V. Ganesh, P. J. Guo, P. Hooimeijer, and M. D. Ernst, "HAMPI: a solver for string constraints," in *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis, ISSTA 2009, Chicago, IL, USA, July 19-23, 2009*, G. Rothermel and L. K. Dillon, Eds. ACM, 2009, pp. 105–116. [Online]. Available: https://doi.org/10.1145/1572272.1572286

[7] N. Bjørner, N. Tillmann, and A. Voronkov, "Path feasibility analysis for string-manipulating programs," in *TACAS*, 2009, pp. 307–321.

[8] R. Amadini, "A survey on string constraint solving," *ACM Comput. Surv.*, vol. 55, no. 2, pp. 16:1–16:38, 2023. [Online]. Available: https://doi.org/10.1145/3484198

[9] N. Rungta, "A billion SMT queries a day (invited paper)," in *Computer Aided Verification - 34th International Conference, CAV 2022, Haifa, Israel, August 7-10, 2022, Proceedings, Part I*, ser. Lecture Notes in Computer Science, S. Shoham and Y. Vizel, Eds., vol. 13371. Springer, 2022, pp. 3–18. [Online]. Available: https://doi.org/10.1007/978-3-031-13185-1_1

[10] J. Backes, P. Bolignano, B. Cook, C. Dodge, A. Gacek, K. S. Luckow, N. Rungta, O. Tkachuk, and C. Varming, "Semantic-based automated reasoning for AWS access policies using SMT," in *2018 Formal Methods in Computer Aided Design, FMCAD 2018, Austin, TX, USA, October 30 - November 2, 2018*, N. S. Bjørner and A. Gurfinkel, Eds. IEEE, 2018, pp. 1–9. [Online]. Available: https://doi.org/10.23919/FMCAD.2018.8602994

[11] C. Barrett, P. Fontaine, and C. Tinelli, "The Satisfiability Modulo Theories Library (SMT-LIB)," `www.SMT-LIB.org`, 2016.

[12] L. de Moura and N. Bjørner, "Z3: an efficient smt solver," in *Tools and Algorithms for the Construction and Analysis of Systems*, vol. 4963, 04 2008, pp. 337–340.

[13] Z. Lu, S. Siemer, P. Jha, J. Day, F. Manea, and V. Ganesh, "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*, K. Larson, Ed. International Joint Conferences on Artificial Intelligence Organization, 8 2024, pp. 1907–1915, main Track. [Online]. Available: https://doi.org/10.24963/ijcai.2024/211

[14] F. Blahoudek, Y.-F. Chen, D. Chocholatý, V. Havlena, L. Holík, O. Lengál, and J. Síč, "Word equations in synergy with regular constraints," in *Formal Methods*, M. Chechik, J.-P. Katoen, and M. Leucker, Eds. Cham: Springer International Publishing, 2023, pp. 403–423.

[15] V. Havlena, L. Holík, O. Lengál, and J. Síč, "Cooking String-Integer Conversions with Noodles," in *27th International Conference on Theory and Applications of Satisfiability Testing (SAT 2024)*, ser. Leibniz International Proceedings in Informatics (LIPIcs), S. Chakraborty and J.-H. R. Jiang, Eds., vol. 305. Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2024, pp. 14:1–14:19. [Online]. Available: https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.SAT.2024.14

[16] Y.-F. Chen, D. Chocholaty, V. Havlena, L. Holik, O. Lengal, and J. Sic, "Solving string constraints with lengths by stabilization," *Proc. ACM Program. Lang.*, vol. 7, no. OOPSLA2, oct 2023. [Online]. Available: https://doi.org/10.1145/3622872

[17] Y.-F. Chen, D. Chocholatý, V. Havlena, L. Holík, O. Lengál, and J. Síč, "Z3-noodler: An automata-based string solver," in *Tools and Algorithms for the Construction and Analysis of Systems*, B. Finkbeiner and L. Kovács, Eds. Cham: Springer Nature Switzerland, 2024, pp. 24–33.

[18] M. Berzish, M. Kulczynski, F. Mora, F. Manea, J. D. Day, D. Nowotka, and V. Ganesh, "An smt solver for regular expressions and linear arithmetic over string length," in *Computer Aided Verification*, A. Silva and K. R. M. Leino, Eds. Cham: Springer International Publishing, 2021, pp. 289–312.

[19] H. Barbosa, C. Barrett, M. Brain, G. Kremer, H. Lachnitt, M. Mann, A. Mohamed, M. Mohamed, A. Niemetz, A. Nötzli, A. Ozdemir, M. Preiner, A. Reynolds, Y. Sheng, C. Tinelli, and Y. Zohar, *cvc5: A Versatile and Industrial-Strength SMT Solver*, 01 2022, pp. 415–442.

[20] F. Mora, M. Berzish, M. Kulczynski, D. Nowotka, and V. Ganesh, "Z3str4: A multi-armed string solver," in *Formal Methods: 24th International Symposium, FM 2021, Virtual Event, November 20–26, 2021, Proceedings*. Berlin, Heidelberg: Springer-Verlag, 2021, p. 389–406. [Online]. Available: https://doi.org/10.1007/978-3-030-90870-6_21

[21] P. A. Abdulla, M. F. Atig, Y.-F. Chen, B. P. Diep, L. Holík, A. Rezine, and P. Rümmer, "Flatten and conquer: a framework for efficient analysis of string constraints," *SIGPLAN Not.*, vol. 52, no. 6, p. 602–617, Jun. 2017. [Online]. Available: https://doi.org/10.1145/3140587.3062384

[22] P. A. Abdulla, M. F. Atig, Y.-F. Chen, B. P. Diep, L. Holik, A. Rezine, and P. Ruemmer, "Trau: Smt solver for string constraints," *2018 Formal Methods in Computer Aided Design (FMCAD)*, pp. 1–5, 2018. [Online]. Available: https://api.semanticscholar.org/CorpusID:53962814

[23] P. A. Abdulla, M. F. Atig, Y.-F. Chen, B. Diep, L. Holík, D. Hu, W.-L. Tsai, Z. Wu, and D.-D. Yen, "Solving not-substring constraint withflat abstraction," in *Programming Languages and Systems*, H. Oh, Ed. Cham: Springer International Publishing, 2021, pp. 305–320.

[24] T. Chen, M. Hague, J. He, D. Hu, A. W. Lin, P. Rümmer, and Z. Wu, "A decision procedure for path feasibility of string manipulating programs with integer data type," in *Automated Technology for Verification and Analysis - 18th International Symposium, ATVA 2020, Hanoi, Vietnam, October 19-23, 2020, Proceedings*, ser. Lecture Notes in Computer Science, D. V. Hung and O. Sokolsky, Eds., vol. 12302. Springer, 2020, pp. 325–342. [Online]. Available: https://doi.org/10.1007/978-3-030-59152-6_18

[25] P. Saxena, D. Akhawe, S. Hanna, F. Mao, S. McCamant, and D. Song, "A symbolic execution framework for javascript," in *2010 IEEE Symposium on Security and Privacy*, 2010, pp. 513–528.

[26] F. Yu, M. Alkhalaf, and T. Bultan, "Stranger: An automata-based string analysis tool for php," 05 2010, pp. 154–157.

[27] M.-T. Trinh, D.-H. Chu, and J. Jaffar, "S3: A symbolic string solver for vulnerability detection in web applications," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '14. New York, NY, USA: Association for Computing Machinery, 2014, p. 1232–1243. [Online]. Available: https://doi.org/10.1145/2660267.2660372

[28] M. Berzish, V. Ganesh, and Y. Zheng, "Z3str3: A string solver with theory-aware heuristics," in *2017 Formal Methods in Computer Aided Design (FMCAD)*, 2017, pp. 55–59.

[29] T. Chen, A. Flores-Lamas, M. Hague, Z. Han, D. Hu, S. Kan, A. W. Lin, P. Rümmer, and Z. Wu, "Solving string constraints with regex-dependent functions through transducers with priorities and variables," *Proc. ACM Program. Lang.*, vol. 6, no. POPL, pp. 1–31, 2022. [Online]. Available: https://doi.org/10.1145/3498707

[30] M. Berglund, F. Drewes, and B. van der Merwe, "Analyzing catastrophic backtracking behavior in practical regular expression matching," in *Proceedings 14th International Conference on Automata and Formal Languages, AFL 2014, Szeged, Hungary, May 27-29, 2014*, ser. EPTCS, Z. Ésik and Z. Fülöp, Eds., vol. 151, 2014, pp. 109–123. [Online]. Available: https://doi.org/10.4204/EPTCS.151.7

[31] M. Berglund and B. van der Merwe, "On the semantics of regular expression parsing in the wild," *Theoretical Computer Science*, vol. 679, pp. 69 – 82, 2017.

[32] J. Harband and K. Smith, "ECMASscript 2020 language specification, 11th edition," 2020, https://262.ecma-international.org/11.0/.

[33] B. Loring, D. Mitchell, and J. Kinder, "Sound regular expression semantics for dynamic symbolic execution of javascript," in *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, Phoenix, AZ, USA, June 22-26, 2019.*, 2019, pp. 425–438. [Online]. Available: https://doi.org/10.1145/3314221.3314645

[34] P. Rümmer, "A constraint sequent calculus for first-order logic with linear integer arithmetic," in *International Conference on Logic for*

Programming Artificial Intelligence and Reasoning. Springer, 2008, pp. 274–289.

[35] A. Møller, "dk.brics.automaton – finite-state automata and regular expressions for java," https://www.brics.dk/automaton, 2021, accessed: 2022-06-23.

[36] H. Hojjat and P. Rümmer, "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*, T. Jebelean, V. Negru, D. Petcu, D. Zaharie, T. Ida, and S. M. Watt, Eds. IEEE Computer Society, 2017, pp. 145–152. [Online]. Available: https://doi.org/10.1109/SYNASC.2017.00033

[37] J. A. Brzozowski, "Derivatives of regular expressions," *J. ACM*, vol. 11, no. 4, p. 481–494, oct 1964. [Online]. Available: https://doi.org/10.1145/321239.321249

[38] B. Eriksson, A. Stjerna, R. De Masellis, P. Rüemmer, and A. Sabelfeld, "Black ostrich: Web application scanning with string solvers," in *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '23. New York, NY, USA: Association for Computing Machinery, 2023, p. 549–563. [Online]. Available: https://doi.org/10.1145/3576915.3616582

[39] A. Lentin, *Equations dans les Monoides Libres*. Gauthier-Villars, Paris, 1972.

[40] V. Diekert, "Makanin's Algorithm," in *Algebraic Combinatorics on Words*, ser. Encyclopedia of Mathematics and its Applications, M. Lothaire, Ed. Cambridge University Press, 2002, vol. 90, ch. 12, pp. 387–442.

[41] P. Backeman, P. Rümmer, and A. Zeljic, "Interpolating bit-vector formulas using uninterpreted predicates and presburger arithmetic," *Formal Methods Syst. Des.*, vol. 57, no. 2, pp. 121–156, 2021. [Online]. Available: https://doi.org/10.1007/s10703-021-00372-6

[42] P. A. Abdulla, M. F. Atig, B. P. Diep, L. Holík, and P. Janků, "Chain-free string constraints," in *Automated Technology for Verification and Analysis*, Y.-F. Chen, C.-H. Cheng, and J. Esparza, Eds. Cham: Springer International Publishing, 2019, pp. 277–293.

Appendix A
Automata Definition

Automaton	::=	`automaton` *Ident* { `init` *State* `;`
		*Tr** `accepting` *State* (`,` *State*)* `;` }
Tr	::=	*State* `->` *State* `[` *Int* `,` *Int* `]` `;`
State	::=	*Ident*
Ident	::=	`[A-Za-z0-9_]+`
Int	::=	`[0-9]+`

Ranges are inclusive and denote Unicode code points as accepted by the current `re.from_automaton` parser (0–65535)[4]. States referenced in transitions are implicitly declared. Whitespace is insignificant.

Appendix B
Transducer Definitions

OSTRICH encodes *transducers* as a collection of mutually recursive Boolean functions in SMT-LIB, where each function corresponds to a distinct state in the transducer. A transducer consists of a set of such states that, at each step, inspect the first character(s) of the input and output strings, select a transition based on guarded conditions, and then recurse on the remaining substrings until termination.

Formally, the behaviour of each state function is structured as follows:

1) Inspect the leading characters of the input and output strings using `str.head`.
2) Evaluate guard conditions to determine the appropriate transition.
3) Invoke another state function (possibly the same one) on the residual substrings obtained via `str.tail`.

Acceptance is defined via *base cases*, typically when both input and output have been fully consumed, expressed as `(= x "")` and `(= y "")`. Character comparisons are performed using Unicode code points, with numeric ranges used to express character classes (e.g., lowercase letters correspond to codes 97–122).

A. Example 1: General Template

In the template below, the transducer is defined by two mutually recursive state functions, `S` and `S2`, each taking as arguments the remaining portions of the input (`x`) and output (`y`) strings. The first clause of each state specifies the *base case*, here accepting when both `x` and `y` are empty. Subsequent clauses correspond to guarded transitions: in `S`, the first transition matches and deletes an `'a'` from the input (consuming only `x` and leaving `y` unchanged), while the second transition copies the current character from input to output (consuming one symbol from each). Each transition invokes the appropriate successor state on the residual strings obtained via `str.tail`. The second state, `S2`, is structurally similar, here illustrated with only a copying transition. This

[4]While OSTRICH2 supports reasoning over the full Unicode range, the automaton parser is presently limited to BMP code points.

structure generalises to arbitrary transducers by adding states, refining guard conditions, and controlling which side(s) of the input–output pair are consumed in each branch.

```
(set-option :parse-transducers true)
; One state per function, (x,y) are the remaining
    ↪ input/output.
(define-funs-rec
  ((S ((x String) (y String)) Bool)
    (S2 ((x String) (y String)) Bool))
  (
   ; S: base case = accept when both consumed
    (or (and (= x "") (= y ""))
   ; transition 1: delete 'a' from input
       (and (not (= x ""))
         (= (str.head x) (char.from-int 97)) ; 'a'
         (S (str.tail x) y))
       ; transition 2: copy char (consume both)
       (and (not (= x "")) (not (= y ""))
         (= (str.head x) (str.head y))
         (S (str.tail x) (str.tail y))))
    ; S2: another state if needed...
    (or (and (= x "") (= y ""))
        (and (not (= x "")) (not (= y ""))
        (= (str.head x) (str.head y))
        (S2 (str.tail x) (str.tail y))))
  )
```

B. Example 2: `toUpper`

The `toUpper` transducer illustrates a length-preserving character transformation. It consists of a single state, `toUpper`, which accepts when both input `x` and output `y` are empty. Otherwise, the guard requires that both strings be non-empty, and the head character of the output is constrained to be either the uppercase equivalent of the head of the input (if the input character is a lowercase letter, identified by Unicode codes 97–122) or an exact copy of the input character in all other cases. This transformation is expressed using the `ite` construct, subtracting 32 from the character code to obtain the uppercase form when applicable. The function then recurses on the tails of both strings, ensuring that the transformation is applied position-wise until the entire input has been processed.

```
(set-option :parse-transducers true)

(define-fun-rec toUpper ((x String) (y String)) Bool
(or (and (= x "") (= y ""))
  (and (not (= x "")) (not (= y ""))
     (= (char.code (str.head y))
       (ite (and (<= 97 (char.code (str.head x)))
                 (<= (char.code (str.head x)) 122))
            (- (char.code (str.head x)) 32)
            (char.code (str.head x))))
     (toUpper (str.tail x) (str.tail y)))))
```

C. Example 3: `extract1st`

The `extract1st` transducer demonstrates a non-length-preserving transformation involving multiple states. Its purpose is to scan the input `x` for the first occurrence of the character `'='` (Unicode code 61), copy the subsequent characters into the output `y` until the next `'='` is encountered, and then skip the remainder of the input. The initial state, `extract1st`, advances through the input without producing output until it finds the first `'='`, at which point it transitions to `extract1st_2`. In this second state, the transducer consumes characters from both input and output in lockstep, copying them directly unless another `'='` is reached, which triggers a transition to `extract1st_3`. The final state, `extract1st_3`, consumes the remaining input without producing further output, thereby terminating the extraction. For example, given the input string `x = "foo=bar"`, the transducer outputs `y = "bar"`.

```
(set-option :parse-transducers true)

(define-funs-rec ((extract1st ((x String) (y String))
    ↪ Bool)
  (extract1st_2 ((x String) (y String)) Bool)
  (extract1st_3 ((x String) (y String)) Bool)) (
  ;
  ; extract1st
  (or (and (= x "") (= y ""))
    (and (not (= x ""))
      (not (= (str.head x) (char.from-int 61)))  ; not
          ↪ '='
      (extract1st (str.tail x) y))
    (and (not (= x ""))
      (= (str.head x) (char.from-int 61))        ; '='
      (extract1st_2 (str.tail x) y)))
  ;
  ; extract1st_2
  (or (and (= x "") (= y ""))
    (and (not (= x "")) (not (= y ""))
      (= (str.head x) (str.head y))
      (not (= (str.head x) (char.from-int 61)))  ; not
          ↪ '='
      (extract1st_2 (str.tail x) (str.tail y)))
    (and (not (= x ""))
      (= (str.head x) (char.from-int 61))        ; '='
      (extract1st_3 (str.tail x) y)))
  ;
  ; extract1st_3
  (or (and (= x "") (= y ""))
    (and (not (= x ""))
      (extract1st_3 (str.tail x) y)))
))
```

D. Grammar for OSTRICH transducers

The following grammar summarises the fragment of SMT-LIB used by OSTRICH for defining such transducers.

```
  Transducer  ::=  (define-fun-rec StateSig Body)
                |  (define-funs-rec (StateSig+) (Body+))

    StateSig  ::=  (Id ((x String) (y String)) Bool)

        Body  ::=  (or Clause+)

      Clause  ::=  BaseCase | Transition

    BaseCase  ::=  (and Guard*)

  Transition  ::=  (and Guard+ Call)

        Call  ::=  (Id XArg YArg)

        XArg  ::=  x | (str.tail x)

        YArg  ::=  y | (str.tail y)

       Guard  ::=  (= Term Term)
                |  (not Guard)
                |  (and Guard+)
                |  (or Guard+)
                |  (<= Int Int)

        Term  ::=  Char | (str.head x) | (str.head y)
                |  (char.code Term)
                |  (ite Guard Term Term)

        Char  ::=  (char.from-int Int)
                |  (str.head x)
                |  (str.head y)

         Int  ::=  Numeral | (char.code Term)
                |  (+ Int Int)
                |  (- Int Int)
```

In this grammar, each `StateSig` corresponds to one *state* in the transducer. The `Body` of a state consists of one or more `Clauses` combined with `or`, where each clause represents either:

- a *base case*, such as `(and (= x "") (= y ""))`, or
- a *transition* guarded by conditions on the heads of `x` and `y` and leading to a recursive `Call` on their tails.

The `Guard` syntax captures conditions such as character equality, inequality, or membership in a Unicode range (as in `toUpper`, which checks lowercase via `(<= 97 (char.code (str.head x)))` and `(<= (char.code (str.head x)) 122)`). `XArg` and `YArg` indicate whether a transition consumes from the input, output, or both. This is the key distinction between length-preserving transformations (both consumed) and non-length-preserving ones (only one consumed at a time).