A DPLL(T) Theory Solver for a Theory of Strings and Regular Expressions*

Tianyi Liang[1], Andrew Reynolds[1], Cesare Tinelli[1], Clark Barrett[2], and Morgan Deters[2]

[1] Department of Computer Science, The University of Iowa
[2] Department of Computer Science, New York University

Abstract. An increasing number of applications in verification and security rely on or could benefit from automatic solvers that can check the satisfiability of constraints over a rich set of data types that includes character strings. Unfortunately, most string solvers today are standalone tools that can reason only about (some fragment) of the theory of strings and regular expressions, sometimes with strong restrictions on the expressiveness of their input language. These solvers are based on reductions to satisfiability problems over other data types, such as bit vectors, or to automata decision problems. We present a set of algebraic techniques for solving constraints over the theory of unbounded strings natively, without reduction to other problems. These techniques can be used to integrate string reasoning into general, multi-theory SMT solvers based on the DPLL(T) architecture. We have implemented them in our SMT solver CVC4 to expand its already large set of built-in theories to a theory of strings with concatenation, length, and membership in regular languages. Our initial experimental results show that, in addition, over pure string problems, CVC4 is highly competitive with specialized string solvers with a comparable input language.

1 Introduction

In the last few years a number of techniques originally developed for verification purposes have been adapted to support software security analyses as well. These techniques have benefited from the rise of powerful specialized reasoning engines such as SMT solvers. Security analyses are frequently required to reason about string values. One reason is that program inputs, especially in web-based applications, are often provided as strings which are then processed using operations such as matching against regular expressions, concatenation, and substring extraction or replacement. In general, both safety and security analyses could benefit from solvers that can check the satisfiability of constraints over a rich set of data types that includes character strings. Despite their power and success as back-end reasoning engines, however, general multi-theory SMT solvers so far have provided minimal or no native support for reasoning over strings.

A major difficulty is that any reasonably comprehensive theory of character strings is undecidable [3]. However, several more restricted, but still quite useful, theories of strings do have a decidable satisfiability problem. These include any theories of fixed-length strings, which are trivially decidable for having a finite domain, but also some

* This work was partially funded by NSF grants #1228765 and #1228768.

fragments over unbounded strings (e.g., word equations [13]). Recent research has focused on identifying decidable fragments suitable for program analysis and, more crucially, on developing efficient solvers for them. Unfortunately, most string solvers today are standalone tools that can reason only about (some fragment of) the theory of strings and regular expressions, sometimes with strong restrictions on the expressiveness of their input language such as, for instance, the imposition of exact length bounds on all string variables. These solvers are based on reductions to satisfiability problems over other data types, such as bit vectors, or to decision problems over automata.

Contribution and significance We present an alternative approach, based on algebraic techniques for solving (quantifier-free) constraints natively over a theory of unbounded strings with length and regular language membership. Our techniques can be used to construct solvers that can be integrated into general, multi-theory SMT solvers based on the DPLL(T) architecture [15]. We have implemented these techniques in our SMT solver CVC4. As as result and to our knowledge, CVC4 is the first solver able to reason about a language of mixed constraints that includes strings together with integers, reals, arrays, and algebraic datatypes. Our experimental results show that, in addition, over pure string problems CVC4 has superior performance and reliability over specialized string solvers that can reason about the same fragment of the theory of strings.

We describe our approach here abstractly in terms of derivation rules. After discussing related work, we define in Section 2 the theory of strings and regular expressions we work with, and present a calculus for this theory. Our string solver is essentially a specific a proof strategy for this calculus. In Section 3, we present an experimental evaluation of our implementation in CVC4 against other tools specializing in string constraints. We conclude in Section 4 mentioning several areas of future work.

1.1 Related work

A popular approach for solving string constraints, especially if they involve regular expressions, is to encode them into automata problems. For example, Hooimeijer and Weimer [9] present an automata-based solver, DPRLE, for matching problems of the form $e \subseteq r$ where, in essence, r is a regular expression over a given alphabet and e is a concatenation of alphabet symbols and string variables. The solver has been used to check programs against SQL injection vulnerabilities. This approach was improved in later work by generating automata lazily from the input problem without requiring *a priori* length bounds [10]. A comprehensive set of algorithms and data structures for performing fast automata operations to support constraint solving over strings is described by Hooimeijer and Veanes [8]. Generally speaking, there are two sorts of automata-based approaches: one where each transition in the automaton represents a single character (e.g., [5, 21]), and one where each transition represents a set of characters (e.g., [10, 19, 20]). Most tools based on these approaches provide very limited support for reasoning about constraints mixing strings and other data types. Also, automata refinement is typically the main bottleneck, although it is still very useful in solving membership constraints. Further discussion can be found in [7, 12].

A different class of solvers is based on reducing string constraints to constraints in other theories. A successful representative of this approach is the Hampi solver [11], used in a variety of static analysis systems. Hampi works only with string constraints over fixed-size string variables. It extends the constraint language to membership in fixed-size context-free languages but considers only problems over one string variable. Input problems are reduced first to bit-vector problems and then to SAT. An alternative approach, developed to support Pex [18], a white-box test generation tool, targets path feasibility problems for programs using the .NET string library [3]. There, string constraints over a large set of string operators, but no language membership predicates, are abstracted to linear integer arithmetic constraints and then sent to an SMT solver. Each satisfying solution, if any, induces a fixed-length version of the original string problem which is then solved using finite domain constraint satisfaction techniques. The Kaluza solver [17] extends Hampi's input language to multiple variables and string concatenation by following an approach similar to one used in Pex, except that it simply feeds fixed-length versions of the input problem to Hampi.

The Java String Analyzer (JSA) [4] works with Java string constraints. It first translates them to a flow graph and then analyzes the graph by converting it into a context-free grammar. That grammar is approximated to a regular one which is then encoded as a multi-level automaton. PASS [12] combines ideas from automata and SMT. Similarly to JSA, it handles almost all Java string operations, regular expressions, and string-number conversions. However, it represents strings as arrays with symbolic length. This leads to the generation of several quantified constraints over such arrays, which are then solved with the aid of a specialized quantifier instantiation procedure.

The work most closely related to ours is Z3-STR [22], a recent string solver developed as an extension of the Z3 SMT solver through Z3's user plug-in interface. It considers unbounded strings with concatenation, substring, replace and length functions and accepts equational constraints over strings as well as linear integer arithmetic constraints. Its main idea is to have Z3 treat string function and predicate symbols as uninterpreted but monitor the inferences of Z3's equality solver and generate and pass to Z3 selected string theory lemmas as needed. Roughly speaking, these lemmas are used to force the identification of equivalent string terms (e.g., the lemma $s \cdot \epsilon \approx s$ where $\cdot$ is concatenation and ϵ is the empty string), or the dis-identification of terms that Z3 has wrongly guessed to be equal (e.g., $\mathsf{len}(t) > 0 \Rightarrow s \not\approx s \cdot t$). The approach is refutationally incomplete because it does not always generate enough axioms to recognize an unsatisfiable problem. At a very high level, our approach is similar, and similarly incomplete, except that it uses a different and more comprehensive set of rules to generate suitable axioms, and so is able to recognize more unsatisfiable cases. Another big difference is that we have devised it with the goal of implementing it in an internal, fully integrated theory solver for CVC4, as opposed to an external plug-in, which allows us to leverage several features of the DPLL(T) architecture.

1.2 Formal preliminaries

We work in the context of many-sorted first-order logic with equality. We assume the reader is familiar with the notions of many-sorted signature, term, literal, formula, free variable, interpretation, and satisfiability of a formula in an interpretation (see, e.g., [2] for more details). A *theory* is a pair $T = (\Sigma, \mathbf{I})$ where Σ is a signature and $\mathbf{I}$ is a class of Σ-interpretations, the *models* of T, that is closed under variable reassignment (i.e., every Σ-interpretation that differs from one in $\mathbf{I}$ only in how it interprets the variables is also in $\mathbf{I}$). If $\mathcal{I}$ is an interpretation and t is a term, we denote by $\mathcal{I}(t)$ the value of t in $\mathcal{I}$. A Σ-formula φ is *satisfiable* (resp., *unsatisfiable*) *in* T if it is satisfied by some (resp., no) interpretation in $\mathbf{I}$. A set Γ of formulas *entails in* T a Σ-formula φ, written $\Gamma \models_T \varphi$, if every interpretation in $\mathbf{I}$ that satisfies all formulas in Γ satisfies φ as well. The set Γ is *satisfiable in* T if $\Gamma \not\models_T \bot$ where $\bot$ is the universally false atom. We will write $\Gamma \models \varphi$ to denote that Γ entails φ in the class of all Σ-interpretations. We will use $\approx$ as the (infix) logical symbol for equality—which has type $\sigma \times \sigma$ for all sorts σ in Σ and is always interpreted as the identity relation. We write $s \not\approx t$ as an abbreviation of $\neg\, s \approx t$. If e is a term or a formula, we denote by $\mathcal{V}(e)$ the set of e's free variables, extending the notation to tuples and sets of terms/formulas as expected. Two Σ-formulas are *equisatisfiable in* T if for every model $\mathcal{A}$ of T that satisfies one there is a model of T that satisfies the other and differs from $\mathcal{A}$ at most over the free variables not shared by the two formulas.

2 A theory of strings and regular language membership

We consider a theory T_{SLRp} of strings with length and positive regular language membership constraints over a signature Σ_{SLRp} with three sorts, Str, Int, and Lan, and an infinite set of variables of each sort. The interpretations of T_{SLRp} differ only on the variables. They all interpret Int as the set of integer numbers, Str as the language $\mathcal{A}^*$ of all words over some fixed finite alphabet $\mathcal{A}$ of *characters*, and Lan as the power set of $\mathcal{A}^*$. The signature includes the following predicate and function symbols: the usual symbols of linear integer arithmetic, interpreted as expected; a constant symbol, or *string constant*, for each word of $\mathcal{A}^*$, interpreted as that word; a variadic function symbol $\mathsf{con} : \mathsf{Str} \times \ldots \times \mathsf{Str} \to \mathsf{Str}$, interpreted as word concatenation; a function symbol $\mathsf{len} : \mathsf{Str} \to \mathsf{Int}$, interpreted as the word length function; a function symbol $\mathsf{set} : \mathsf{Str} \to \mathsf{Lan}$, interpreted as the function mapping each word $w \in \mathcal{A}^*$ to the language $\{w\}$; a function symbol $\mathsf{star} : \mathsf{Lan} \to \mathsf{Lan}$, interpreted as the Kleene closure operator; an infix predicate symbol $\mathsf{in} : \mathsf{Str} \times \mathsf{Lan}$, interpreted as the set membership predicate; a suitable set of additional function symbols corresponding to regular expression operators such as language concatenation, conjunction, disjunction, and so on.

We call: *string term* any term of sort Str or of the form $(\mathsf{len}\, s)$; *arithmetic term* any term of sort Int all of whose occurrences of len are applied to a variable; *regular expression* any term of sort Lan (possibly with variables). A string term is *atomic* if it is a variable or a string constant. A *string constraint* is a (dis)equality $(\neg)s \approx t$ with s and t string terms. What algebraists call *word equations* are, in our terminology, positive string constraints $s \approx t$ with s and t of sort Str. An *arithmetic constraint* is a (dis)equality $(\neg)s \approx t$ or an inequality $s > t$ where s and t are arithmetic terms. Note that if x and y are string variables, $\mathsf{len}\, x$ is both a string and an arithmetic term and $(\neg)\mathsf{len}\, x \approx \mathsf{len}\, y$ is both a string and an arithmetic constraint. A *(positive) RL constraint* is a literal of the form $(s \;\mathsf{in}\; r)$ where s is a string term and r is a regular expression. A T_{SLRp}-*constraint* is a string, arithmetic or RL constraint. We will denote entailment in T_{SLRp} ($\models_{\mathsf{SLRp}}$) more simply as $\models_{\mathsf{SLRp}}$.

2.1 The satisfiability problem in T_{SLRp}

We are interested in checking the satisfiability in T_{SLRp} of finite sets of T_{SLRp}-constraints. We are not aware of any results on the decidability of this problem. In fact, the decidability of a strict sublanguage of the above, just word equations with length constraints, is classified as an open question by other authors (e.g., [6]). Some other sublanguages do have a decidable satisfiability problem. For instance, the satisfiability of word equations was proven decidable by Makanin [13] and then given a PSPACE algorithm by Plandowski [16]; that algorithm, however, is highly impractical.

In this work we focus on practical solvers for T_{SLRp} that, although incomplete and non-terminating in general, can be used to solve efficiently string constraints arising from verification and security applications. In addition to efficiency, we also strive for correctness. We want a solver that is both *refutation sound*: any problem the solver classifies as unsatisfiable is indeed so; and *solution sound*: any variable assignment that the solver claims to be a solution of the input constraints does indeed satisfy them.

Our solver is based on the modular combination of an off-the-shelf solver for linear integer arithmetic and a novel solver for string and RL constraints, which we will call just string solver, for brevity. The string solver is in turn obtained as a modular extension of a congruence-closure-based solver for EUF, the theory of equality with uninterpreted functions. The extension is obtained by means of theory-specific derivation rules that assert additional string constraints and RL constraints to the congruence closure module (which treats all functions symbols as uninterpreted). The combination between the string solver and the arithmetic solver is achieved, Nelson-Oppen style, by exchanging equalities over shared terms, which however are not variables, as in traditional combination procedures [14], but terms of the form $(\mathsf{len}\, x)$ where x is a variable.[3]

In the following, we describe the essence of our combined solver for T_{SLRp} abstractly and declaratively, as a tableaux-style calculus. Because of the computational complexity of solving even just word equations, this calculus is non-deterministic and allows many possible proof strategies. Our solver can be understood then as a specific proof procedure for the calculus. In our description below we focus only on the derivation rules that deal with string and arithmetic constraints. This is both because of space constraints and because currently our treatment of RL constraints is fairly naive—and so not very interesting. In particular, the Kleene star operator is processed by unrolling: $(s \;\mathsf{in}\; \mathsf{star}\; r)$ is reduced to $s = \epsilon$ or to $s \approx \mathsf{con}(x, y) \wedge (x \;\mathsf{in}\; r) \wedge (y \;\mathsf{in}\; \mathsf{star}\; r)$ where x and y are fresh variables, which makes the solver non-terminating in general over such constraints. A more sophisticated treatment of RL constraints is in the works and will be presented in a later paper.

2.2 A calculus for T_{SLRp}

Let S be a set of string constraints and let $\mathcal{T}(S)$ be the set of all terms (and subterms) occurring in S. The *congruence closure* of S is the set

$$\begin{aligned}\mathcal{C}(S) = \{s \approx t \mid s, t \in \mathcal{T}(S),\, S \models s \approx t\} \cup \{l_1 \not\approx l_2 \mid l_1, l_2 \text{ distinct string const.}\} \cup \\ \{s \not\approx t \mid s, t \in \mathcal{T}(S),\, s' \not\approx t' \in S,\, S \models s \approx s' \wedge t \approx t' \text{ for some } s', t'\}\end{aligned}$$

[3] This difference is not substantial if the arithmetic solver treats $(\mathsf{len}\, x)$ like an integer variable.

$$
\begin{aligned}
\mathsf{con}(\boldsymbol{s}, \mathsf{con}(\boldsymbol{t}), \boldsymbol{u}) &\to \mathsf{con}(\boldsymbol{s}, \boldsymbol{t}, \boldsymbol{u}) & \mathsf{con}(\boldsymbol{s}, c_1 \cdots c_i, c_{i+1} \cdots c_n, \boldsymbol{u}) &\to \mathsf{con}(\boldsymbol{s}, c_1 \cdots c_n, \boldsymbol{u}) \\
\mathsf{con}(\boldsymbol{s}, \epsilon, \boldsymbol{u}) &\to \mathsf{con}(\boldsymbol{s}, \boldsymbol{u}) & \mathsf{len}(\mathsf{con}(s_1, \ldots, s_n)) &\to \mathsf{len}\, s_1 + \cdots + \mathsf{len}\, s_n \\
\mathsf{con}(s) &\to s & \mathsf{len}(c_1 \cdots c_n) &\to n \\
\mathsf{con}() &\to \epsilon
\end{aligned}
$$

Fig. 1. Normalization rewrite rules for terms.

The set $\mathcal{C}(S)$ induces an equivalence relation $\mathbf{E}_S$ over $\mathcal{T}(S)$ where two terms s, t are equivalent iff $s \approx t \in \mathcal{C}(S)$ (or, equivalently, iff $S \models s \approx t$). For all $t \in \mathcal{T}(S)$, we denote its equivalence class in $\mathbf{E}_S$ by $[t]_S$ or just $[t]$ when S is clear or not important.

We will denote characters (i.e., elements of the alphabet $\mathcal{A}$) by the letter c and string constants by l or the juxtaposition $c_1 \cdots c_n$ of their individual characters, with $c_1 \cdots c_n$ denoting the empty string ϵ when $n = 0$. We will use x, y, z to denote string variables and s, t, u, v, w to denote terms in general.

We will consider term tuples $(s_1, \ldots, s_n)$, with $n \geq 0$, and denote them by letters in bold font, with comma denoting tuple concatenation. For example, if $\boldsymbol{s} = (s_1, s_2)$ and $\boldsymbol{t} = (t_1, t_2, t_3)$ we will write $(\boldsymbol{s}, \boldsymbol{t})$ to denote the tuple $(s_1, s_2, t_1, t_2, t_3)$. Similarly, if u is a term, $(\boldsymbol{s}, u, \boldsymbol{t})$ denotes the tuple $(s_1, s_2, u, t_1, t_2, t_3)$.

Configurations Our calculus operates over *configurations* consisting of the distinguished configuration unsat and of tuples of the form $\langle \mathsf{S}, \mathsf{A}, \mathsf{R}, \mathsf{F}, \mathsf{N}, \mathsf{C}, \mathsf{B} \rangle$ where

- $\mathsf{S}, \mathsf{A}, \mathsf{R}$ are respectively a set of string, arithmetic, and RL constraints;
- F is a set of pairs $s \mapsto \boldsymbol{a}$ where $s \in \mathcal{T}(\mathsf{S})$ and $\boldsymbol{a}$ is a tuple of atomic string terms;
- N is a set of pairs $e \mapsto \boldsymbol{a}$ where e is an equivalence class of $\mathbf{E}_{\mathsf{S}}$, the equivalence relation induced by the constraints in S, and $\boldsymbol{a}$ is a tuple of atomic string terms;
- C is a set of terms of sort Str;
- B is a set of *buckets* where each bucket is a set of equivalence classes of $\mathbf{E}_{\mathsf{S}}$.

Informally, the sets $\mathsf{S}, \mathsf{A}, \mathsf{R}$ initially store the input problem and grow with additional constraints derived by the calculus; N stores a normal form for each equivalence class in $\mathbf{E}_{\mathsf{S}}$; F maps selected input terms to an intermediate form, which we call a *flat form*, used to compute the normal forms in N; C stores terms whose flat form should not be computed, to prevent loops in the computation of their equivalence class' normal form; B eventually becomes a partition of $\mathbf{E}_{\mathsf{S}}$ used to generate a satisfying assignment that assigns string constants of different lengths to variables in different buckets, and different string constants of the *same* length to different variables in the same bucket.

Derivation trees The calculus is defined by the derivation rules described below. A *derivation tree* for the calculus is a tree where each node is a configuration and each non-root node is obtained by applying one of the derivation rules to its parent node. We call the root of a derivation tree an *initial* configuration. A branch of a derivation tree is *closed* if it ends with unsat. A derivation tree is *closed* if all of its branches are closed.

Initial configurations encode a satisfiability problem by storing it in the components S, A and R. By standard transformations, one can convert any finite set of T_{SLRp}-constraints into an equisatisfiable set $S \cup A \cup R$ where S is a set of string constraints, A is a set of arithmetic constraints, and R is a set of RL constraints. We consider only initial configurations where the other components are empty. For convenience, we assume

$$\textsf{A-Prop}\ \frac{\mathsf{S} \models \mathsf{len}\,x \approx \mathsf{len}\,y}{\mathsf{A} := \mathsf{A}, \mathsf{len}\,x \approx \mathsf{len}\,y} \qquad \textsf{S-Prop}\ \frac{\mathsf{A} \models_{\mathsf{LIA}} \mathsf{len}\,x \approx \mathsf{len}\,y}{\mathsf{S} := \mathsf{S}, \mathsf{len}\,x \approx \mathsf{len}\,y}$$

$$\textsf{Len}\ \frac{x \approx t \in \mathcal{C}(\mathsf{S}) \quad x \in \mathcal{V}(\mathsf{S})}{\mathsf{A} := \mathsf{A}, \mathsf{len}\,x \approx (\mathsf{len}\,t){\downarrow}} \qquad \textsf{Len-Split}\ \frac{x \in \mathcal{V}(\mathsf{S} \cup \mathsf{A}) \quad x : \mathsf{Str}}{\mathsf{S} := \mathsf{S}, x \approx \epsilon \quad \| \quad \mathsf{A} := \mathsf{A}, \mathsf{len}\,x > 0}$$

$$\textsf{A-Conflict}\ \frac{\mathsf{A} \models_{\mathsf{LIA}} \bot}{\mathsf{unsat}} \qquad \textsf{R-Star}\ \frac{s \ \mathsf{in}\ \mathsf{star}(\mathsf{set}\ t) \in \mathsf{R} \quad s \not\approx \epsilon \in \mathcal{C}(\mathsf{S})}{\mathsf{S} := \mathsf{S}, s \approx \mathsf{con}(t, z) \quad \mathsf{R} := \mathsf{R}, z \ \mathsf{in}\ \mathsf{star}(\mathsf{set}\ t)}$$

Fig. 2. Rules for theory combination, arithmetic and RL constraints. The letter z denotes a fresh Skolem variable.

that the S component of the initial configuration contains an equation $x \approx t$ for each non-variable term $t \in \mathcal{T}(\mathsf{S})$, where x is a variable of the same sort as t.[4] We also assume that all terms in the initial configuration are reduced with respect to the rewrite rules in Figure 1, which can be shown to be terminating and confluent modulo the axioms of arithmetic.

We say that a configuration is *derivable* if it occurs in a derivation tree whose initial configuration satisfies the restrictions above.

We denote by $t{\downarrow}$ the normal form of a term t with respect to the rewrite rules in Figure 1. It is not difficult to see that if t is of sort Str, then $t{\downarrow}$ is either an atomic string term or has the form $\mathsf{con}(a_1, \ldots, a_n)$ where $n > 1$ and $a_1, \ldots, a_n$ are atomic; if t is of integer sort, then $t{\downarrow}$ is an arithmetic term. In a similar vein, we consider *normalized* tuples $\boldsymbol{a}{\downarrow}$ of atomic terms obtained from an atomic term tuple $\boldsymbol{a}$ by dropping its empty string components and replacing adjacent string constants by the constant corresponding to their concatenation. For example, $(x, \epsilon, c_1, c_2c_3, y){\downarrow} = (x, c_1c_2c_3, y)$.

Invariant 1 We are interested in proof procedures that maintain these invariants on the derivable configurations of the form $\langle \mathsf{S}, \mathsf{A}, \mathsf{R}, \mathsf{F}, \mathsf{N}, \mathsf{C}, \mathsf{B} \rangle$:

1. All terms are reduced with respect to the rewrite system in Figure 1.
2. F is a partial map from $\mathcal{T}(\mathsf{S})$ to normalized tuples of atomic terms.
3. N is a partial map from $\mathbf{E}_\mathsf{S}$ to normalized tuples of atomic terms.
4. For all terms s where $[s] \mapsto (a_1, \ldots, a_n) \in \mathsf{N}$ or $s \mapsto (a_1, \ldots, a_n) \in \mathsf{F}$, we have $\mathsf{S} \models_{\mathsf{SLRp}} s \approx \mathsf{con}(a_1, \ldots, a_n)$ and $\mathsf{S} \models a_i \not\approx \epsilon$ for $i = 1, \ldots, n$.
5. For all $B_1, B_2 \in \mathsf{B}$, $[s] \in B_1$ and $[t] \in B_2$, $\mathsf{S} \models \mathsf{len}\,s \approx \mathsf{len}\,t$ iff $B_1 = B_2$.
6. C contains only reduced terms of the form $\mathsf{con}(\boldsymbol{a})$.

We denote by $\mathcal{D}(\mathsf{N})$ the *domain* of the partial map N, i.e., the set $\{e \mid e \mapsto \boldsymbol{a} \in \mathsf{N}$ for some $\boldsymbol{a}\}$. For all $e \in \mathcal{D}(\mathsf{N})$, we will write $\mathsf{N}\,e$ to denote the (unique) tuple associated to e by N. We will use a similar notation for F.

Derivation rules The rules of the calculus are provided in Figures 2 through 6 in *guarded assignment form*. A derivation rule applies to a configuration K if all of the rule's premises hold for K. A rule's conclusion describes how each component of K is

[4] Such equations can always be added as needed using fresh variables.

$$\text{S-Cycle}\ \frac{t = \mathsf{con}(t_1, \ldots, t_i, \ldots, t_n) \quad t \in \mathcal{T}(\mathsf{S}) \setminus \mathsf{C} \qquad t_k \approx \epsilon \in \mathcal{C}(\mathsf{S}) \text{ for all } k \in \{1, \ldots, n\} \setminus \{i\}}{\mathsf{S} := \mathsf{S}, t \approx t_i \quad \mathsf{C} := (\mathsf{C}, t) \setminus \{t_i\}} \qquad \text{Reset}\ \frac{}{\mathsf{F} := \emptyset \quad \mathsf{N} := \emptyset \quad \mathsf{B} := \emptyset}$$

$$\text{S-Split}\ \frac{x, y \in \mathcal{V}(\mathsf{S}) \quad x \approx y,\, x \not\approx y \notin \mathcal{C}(\mathsf{S})}{\mathsf{S} := \mathsf{S}, x \approx y \quad \| \quad \mathsf{S} := \mathsf{S}, x \not\approx y} \qquad \text{S-Conflict}\ \frac{s \approx t \in \mathcal{C}(\mathsf{S}) \quad s \not\approx t \in \mathcal{C}(\mathsf{S})}{\mathsf{unsat}}$$

$$\text{L-Split}\ \frac{x, y \in \mathcal{V}(\mathsf{S}) \quad x, y : \mathsf{Str} \quad \mathsf{S} \not\models \mathsf{len}\, x \approx \mathsf{len}\, y \quad \mathsf{S} \not\models \mathsf{len}\, x \not\approx \mathsf{len}\, y}{\mathsf{S} := \mathsf{S}, \mathsf{len}\, x \approx \mathsf{len}\, y \quad \| \quad \mathsf{S} := \mathsf{S}, \mathsf{len}\, x \not\approx \mathsf{len}\, y}$$

Fig. 3. Basic string derivation rules.

changed, if at all. We write S, t as an abbreviation for $S \cup \{t\}$. Rules with two conclusions, separated by the symbol $\|$, are non-deterministic branching rules.

In the rules of the calculus, we treat a string constant l in a tuple of terms indifferently as term or a tuple $l_1, \ldots, l_n$ of string constants whose concatenation equals l. For example, a tuple $(x, c_1c_2c_3, y)$ with the three-character constant $c_1c_2c_3$ will be seen also as the tuple (x, c_1, c_2c_3, y), (x, c_1c_2, c_3, y), or (x, c_1, c_2, c_3, y). All equalities and disequalities in the rules are treated modulo symmetry of $\approx$. We assume the availability of a procedure for checking entailment in the theory of linear integer arithmetic ($\models_{\mathsf{LIA}}$) and one for computing congruence closures and checking entailment in EUF ($\models$).

The first four rules in Figure 2 describe the interaction between arithmetic reasoning and string reasoning, achieved via the propagation of entailed constraints in the shared language. R-Star is the only rule for handling RL constraints that we provide here. We chose it because the constraints matching its premise can be generated, by rule F-Loop in Figure 5, even if the initial configuration contains no RL constraints. The basic rules for string constraints are shown in Figure 3. The functionality and rationale of the last three should be straightforward. Reset is meant to be applied after the set S changes since in that case normal and flat forms may need updating. S-Cycle identifies a concatenation of terms with one them when the remaining ones are all equivalent to ϵ.

The bulk of the work is done by the rules in Figures 4 and 5. Those in Figure 4 compute an equivalent flat form (consisting of a sequence of atomic terms) for all non-variable terms that are not in the set C. Flat forms are used in turn to compute normal forms as follows. When all terms of an equivalence class e except for variables and terms in C have the same flat form, that form is chosen by N-Form1 as the normal form of e. When an equivalence class e consists only of variables and terms in C, one of them is chosen by N-Form2 as the normal form of e. The first two rules of Figure 5 use flat forms to add to S new equations entailed by S in the theory of strings. F-Loop is used to recognize and break certain occurrences of reasoning *loops* that lead to infinite paths in a derivation tree (see [**?**] for more details).

The rules in Figure 6 are used to put equivalence classes of terms of sort Str into buckets based on the expected length of the value they will be given eventually by a satisfying assignment. The main idea is that different equivalence classes go into different buckets (using D-Base) unless they have the same length. In the latter case, they go into the same bucket only if we can tell they cannot have the same value (using

$$\text{F-Form1}\ \frac{t = \mathsf{con}(t_1,\ldots,t_n) \quad t \in \mathcal{T}(\mathsf{S}) \setminus (\mathcal{D}(\mathsf{F}) \cup \mathsf{C}) \quad \mathsf{N}\,[t_1] = \boldsymbol{s}_1 \ \cdots\ \mathsf{N}\,[t_n] = \boldsymbol{s}_n}{\mathsf{F} := \mathsf{F}, t \mapsto (\boldsymbol{s}_1,\ldots,\boldsymbol{s}_n)\!\downarrow} \qquad \text{F-Form2}\ \frac{l \in \mathcal{T}(\mathsf{S}) \setminus \mathcal{D}(\mathsf{F})}{\mathsf{F} := \mathsf{F}, l \mapsto (l)}$$

$$\text{N-Form1}\ \frac{[x] \notin \mathcal{D}(\mathsf{N}) \quad s \in [x] \setminus (\mathsf{C} \cup \mathcal{V}(\mathsf{S})) \quad \mathsf{F}\,t = \mathsf{F}\,s \ \text{ for all }\ t \in [x] \setminus (\mathsf{C} \cup \mathcal{V}(\mathsf{S}))}{\mathsf{N} := \mathsf{N}, [x] \mapsto \mathsf{F}\,s} \qquad \text{N-Form2}\ \frac{[x] \notin \mathcal{D}(\mathsf{N}) \quad [x] \subseteq \mathsf{C} \cup \mathcal{V}(\mathsf{S})}{\mathsf{N} := \mathsf{N}, [x] \mapsto (x)}$$

Fig. 4. Normalization derivation rules. The letter l denotes a string constant.

$$\text{F-Unify}\ \frac{\mathsf{F}\,s = (\boldsymbol{w}, u, \boldsymbol{u}_1) \quad \mathsf{F}\,t = (\boldsymbol{w}, v, \boldsymbol{v}_1) \quad s \approx t \in \mathcal{C}(\mathsf{S}) \quad \mathsf{S} \models \mathsf{len}\,u \approx \mathsf{len}\,v}{\mathsf{S} := \mathsf{S}, u \approx v}$$

$$\text{F-Split}\ \frac{\mathsf{F}\,s = (\boldsymbol{w}, u, \boldsymbol{u}_1) \quad \mathsf{F}\,t = (\boldsymbol{w}, v, \boldsymbol{v}_1) \quad s \approx t \in \mathcal{C}(\mathsf{S}) \quad \mathsf{S} \models \mathsf{len}\,u \not\approx \mathsf{len}\,v \quad u \notin \mathcal{V}(\boldsymbol{v}_1) \quad v \notin \mathcal{V}(\boldsymbol{u}_1)}{\mathsf{S} := \mathsf{S}, u \approx \mathsf{con}(v, z) \quad \| \quad \mathsf{S} := \mathsf{S}, v \approx \mathsf{con}(u, z)}$$

$$\text{F-Loop}\ \frac{\mathsf{F}\,s = (\boldsymbol{w}, x, \boldsymbol{u}_1) \quad \mathsf{F}\,t = (\boldsymbol{w}, v, \boldsymbol{v}_1, x, \boldsymbol{v}_2) \quad s \approx t \in \mathcal{C}(\mathsf{S}) \quad x \notin \mathcal{V}((v, \boldsymbol{v}_1))}{\begin{array}{c}\mathsf{S} := \mathsf{S},\ x \approx \mathsf{con}(z_2, z),\ \mathsf{con}(v, \boldsymbol{v}_1) \approx \mathsf{con}(z_2, z_1),\ \mathsf{con}(\boldsymbol{u}_1) \approx \mathsf{con}(z_1, z_2, \boldsymbol{v}_2) \\ \mathsf{R} := \mathsf{R}, z \ \mathsf{in}\ \mathsf{star}(\mathsf{set}\ \mathsf{con}(z_1, z_2)) \quad \mathsf{C} := \mathsf{C}, t\end{array}}$$

Fig. 5. Equality reduction rules. The letters z, z_1, z_2 denote fresh Skolem variables.

D-Add). D-Split is used to reduce the problem to one of the two previous cases. The goal is that, on saturation, each bucket B can be assigned a unique length n_B, and each equivalence class in B can evaluate to a unique string constant of that length. Card makes sure that n_B is big enough to have enough string constants of length n_B.

Correctness We now formalize the main correctness properties of our calculus. Since our solver can be seen as a specific proof procedure, it immediately inherits those properties. This means in particular that when our solver terminates with a sat or unsat answer, that answer is correct. We describe here only the more restricted case of input problems with no RL constraints, as those constraints are not the focus of this work. Also, we consider only derivation trees satisfying Invariant 1. We start with the following lemmas.

Lemma 1. *For all terms t of sort* Str, $\models_{\mathsf{SLRp}} t \approx t\!\downarrow$.

Proof. Immediate consequence of the fact that in each of the rewrite rules of Figure 1, the left-hand side is equivalent in T_{SLRp} to the right hand side.

Lemma 2. *Invariant 1 holds for all derivable configurations.*

Proof. First, Invariant 1 trivially holds for any initial configuration. Thus, we show that parts 1 through 5 of Invariant 1 are preserved for each rule application in a derivation tree.

Part 1 is preserved by noting that all rules (such as F-Split, F-Loop, D-Split) only introduce new terms that are normalized with respect to Figure 1. Parts 2 and 3 are

$$\text{D-Base}\ \frac{s \in \mathcal{T}(\mathsf{S}) \quad s : \mathsf{Str} \qquad S \models \mathsf{len}\, s \approx len_B \text{ for no } B \in \mathsf{B}}{\mathsf{B} := \mathsf{B}, \{[s]\}} \qquad \text{Card}\ \frac{B \in \mathsf{B} \quad |B| > 1}{\mathsf{A} := \mathsf{A}, len_B > \lfloor \log_{|\mathcal{A}|}(|B| - 1) \rfloor}$$

$$\text{D-Add}\ \frac{\begin{array}{c} s \in \mathcal{T}(\mathsf{S}) \quad s : \mathsf{Str} \quad \mathsf{B} = \mathsf{B}', B \quad S \models \mathsf{len}\, s \approx len_B \quad [s] \notin B \\ \text{for all } e \in B \text{ there are } \boldsymbol{w}, u, \boldsymbol{u}_1, v, \boldsymbol{v}_1 \text{ such that} \\ (\mathsf{N}\,[s] = (\boldsymbol{w}, u, \boldsymbol{u}_1),\ \mathsf{N}\, e = (\boldsymbol{w}, v, \boldsymbol{v}_1),\ S \models \mathsf{len}\, u \approx \mathsf{len}\, v,\ u \not\approx v \in \mathcal{C}(\mathsf{S})) \end{array}}{\mathsf{B} := \mathsf{B}', (B \cup \{[s]\})}$$

$$\text{D-Split}\ \frac{\begin{array}{c} s \in \mathcal{T}(\mathsf{S}) \quad s : \mathsf{Str} \quad \mathsf{B} = \mathsf{B}', B \quad S \models \mathsf{len}\, s \approx len_B \quad [s] \notin B \quad e \in B \\ \mathsf{N}\,[s] = (\boldsymbol{w}, u, \boldsymbol{u}_1) \quad \mathsf{N}\, e = (\boldsymbol{w}, v, \boldsymbol{v}_1) \quad S \models \mathsf{len}\, u \not\approx \mathsf{len}\, v \end{array}}{\mathsf{S} := \mathsf{S}, u \approx \mathsf{con}(z_1, z_2), \mathsf{len}\, z_1 \approx \mathsf{len}\, v \quad \| \quad \mathsf{S} := \mathsf{S}, v \approx \mathsf{con}(z_1, z_2), \mathsf{len}\, z_1 \approx \mathsf{len}\, u}$$

Fig. 6. Disequality reduction rules. Letters z_1, z_2 denote fresh Skolem variables. For each bucket $B \in \mathsf{B}$, len_B denotes a unique term $(\mathsf{len}\, x)$ where $[x] \in B$. $|\_|$ denotes the cardinality operator.

preserved by noting that the premises in Figure 4 ensure that entries can be added to F only for terms from $\mathcal{T}(\mathsf{S})$ not in the domain of F, and similarly for N.

The rules in Figure 4 preserve part 4. To show it is preserved for F-Form1, since by assumption of part 4 of the invariant on the premises we have that $\mathsf{S} \models_{\mathsf{SLRp}} t_1 \approx \mathsf{con}(\boldsymbol{s}_1) \wedge \ldots \wedge t_n \approx \mathsf{con}(\boldsymbol{s}_n)$. Since con is associative and due to Lemma 1, we have that $\mathsf{S} \models_{\mathsf{SLRp}} t \approx \mathsf{con}(\boldsymbol{s}_1, \ldots, \boldsymbol{s}_n){\downarrow}$. To show it is preserved for F-Form2, note that $\mathsf{S} \models_{\mathsf{SLRp}} l \approx \mathsf{con}(l){\downarrow}$ for any l, and $(l){\downarrow}$ is either the empty tuple or a tuple containing a non-empty string constant. To show it is preserved for N-Form1, by assumption of the invariant on the premises we have $\mathsf{S} \models_{\mathsf{SLRp}} s \approx \mathsf{con}(\mathsf{F}s)$, and since $x \approx s \in \mathcal{C}(S)$, we have $\mathsf{S} \models_{\mathsf{SLRp}} x \approx \mathsf{con}(\mathsf{F}s)$. Finally, to show it is preserved for N-Form2, note that $\mathsf{S} \models_{\mathsf{SLRp}} x \approx \mathsf{con}(x)$. Furthermore $\mathsf{S} \not\models x \approx \epsilon$, since if this was the case, then $[x]$ would contain the term ϵ, which is not in $\mathsf{C} \cup \mathcal{V}(\mathsf{S})$ by assumption of part 6 of the invariant.

Part 5 is preserved by the rule D-Base, which creates a new bucket containing equivalence class $[s]$ only when $\mathsf{S} \not\models \mathsf{len}\, s \approx len_B$ for any $B \in \mathsf{B}$. It is also preserved by D-Add, which only adds equivalence classes $[s]$ to buckets B when $\mathsf{len}\, s$ is equal to len_B, which by assumption of Invariant 1 part 5 on all other $[t] \in B$ and transitivity of equality we have that $\mathsf{S} \models \mathsf{len}\, s \approx \mathsf{len}\, t$.

Finally, part 6 is preserved by S-Cycle. Applying F-Loop also preserves part 6, by noting that $\mathsf{F}\, t$ contains a variable x, and thus must have been constructed from an application of F-Form1, implying that t is a term of the form $\mathsf{con}(t_1, \ldots, t_n)$. □

Theorem 1 (Refutation Soundness) *For all closed derivation trees with initial configuration $\langle S_0, A_0, \emptyset, \emptyset, \emptyset, \emptyset, \emptyset \rangle$, the set $S_0 \cup A_0$ is unsatisfiable in T_{SLRp}.*

Proof. Assume that $\langle S_0, A_0, R_0, \emptyset, \emptyset, \emptyset, \emptyset \rangle$ has a closed derivation tree D. We show the theorem by induction on the size of the derivation tree for all nodes $c = \langle \mathsf{S}, \mathsf{A}, \mathsf{R}, \mathsf{F}, \mathsf{N}, \mathsf{C}, \mathsf{B} \rangle$ in D.

First, if the derivation tree is an application of S-Conflict or A-Conflict, then $\mathsf{S} \cup \mathsf{A} \cup \mathsf{R}$ is unsatisfiable in T_{SLRp}.

If the children of c are obtained by an application of Len-Split, S-Split, and L-Split, then each child is the root of a closed derivation tree. Since all models satisfy exactly one of the two branches, by the induction hypothesis on both children, we have that $\mathsf{S} \cup \mathsf{A} \cup \mathsf{R}$ is unsatisfiable in T_{SLRp}.

If the child of c is an application of Len, R-Star, A-Prop, S-Prop, S-Cycle, F-Unify, F-Loop, or a rule that does not modify S, A, or R, then it is a configuration of the form $\langle \mathsf{S}', \mathsf{A}', \mathsf{R}', \mathsf{F}', \mathsf{N}', \mathsf{C}', \mathsf{B}' \rangle$ where $\mathsf{S}' \cup \mathsf{A}' \cup \mathsf{R}'$ is equisatisfiable modulo T_{SLRp} to $\mathsf{S} \cup \mathsf{A} \cup \mathsf{R}$. For Len, A-Prop, and S-Prop, this is immediate. For R-Star, this holds since s is not empty and s is in $\mathsf{star}(\mathsf{set}(t))$, and thus s must be the concatenation of one or more copies of t. For S-Cycle, notice that $\mathsf{S} \models_{\mathsf{SLRp}} t \approx \mathsf{con}(\epsilon, \ldots, t_i, \ldots, \epsilon)$, and hence $\mathsf{S} \models_{\mathsf{SLRp}} t \approx t_i$. For F-Unify, we have that by part 4 of Invariant 1 for s and t, $\mathsf{S} \models_{\mathsf{SLRp}} s \approx \mathsf{con}(\mathsf{F}\, s) \wedge t \approx \mathsf{con}(\mathsf{F}\, t)$, and since $s \approx t \in \mathcal{C}(\mathsf{S})$, we have that $\mathsf{S} \models_{\mathsf{SLRp}} \mathsf{con}(\mathsf{F}\, s) \approx \mathsf{con}(\mathsf{F}\, t)$. Thus, since $\mathsf{S} \models \mathsf{len}\, u \approx \mathsf{len}\, v$, we have that $\mathsf{S} \models_{\mathsf{SLRp}} u \approx v$. For F-Loop, we refer to Lemma 5 in the Appendix. In each of these cases, by the induction hypothesis, since $\langle \mathsf{S}', \mathsf{A}', \mathsf{R}', \mathsf{F}', \mathsf{N}', \mathsf{C}', \mathsf{B}' \rangle$ is the root of a closed derivation tree, $\mathsf{S}' \cup \mathsf{A}' \cup \mathsf{R}'$ is unsatisfiable in T_{SLRp}, and thus we have that $\mathsf{S} \cup \mathsf{A} \cup \mathsf{R}$ is unsatisfiable in T_{SLRp}.

If the children of c are obtained by an application of F-Split, then each child is the root of a closed derivation tree. By part 4 of Invariant 1 for s and t, we have that $\mathsf{S} \models_{\mathsf{SLRp}} s \approx \mathsf{con}(\mathsf{F}\, s) \wedge t \approx \mathsf{con}(\mathsf{F}\, t)$, and since $s \approx t \in \mathcal{C}(S)$, we have that $\mathsf{S} \models_{\mathsf{SLRp}} \mathsf{con}(\mathsf{F}\, s) \approx \mathsf{con}(\mathsf{F}\, t)$. Since the lengths of u and v are entailed to be disequal by S, in all models of $\mathsf{S} \cup \mathsf{A} \cup \mathsf{R}$, we have that u is a prefix of v or vice versa. Assume there is a model $\mathcal{M}$ of $\mathsf{S} \cup \mathsf{A} \cup \mathsf{R}$ where u is a prefix of v, say $\mathcal{M}(u) = l_1$ and $\mathcal{M}(v) = l_1 l_2$. Consider an extension of this model $\mathcal{M}'$ such that $\mathcal{M}'(k) = l_2$, and $\mathcal{M}'(x) = \mathcal{M}(x)$ for all other variables x. We have that $\mathcal{M}'$ is a model for $\mathsf{S} \cup \{u \approx \mathsf{con}(v, k)\} \cup \mathsf{A} \cup \mathsf{R}$. However, by the induction hypothesis on the left child, we have that $\mathsf{S} \cup \{u \approx \mathsf{con}(v, k)\} \cup \mathsf{A} \cup \mathsf{R}$ is unsatisfiable in T_{SLRp}. Thus, there are no models of $\mathsf{S} \cup \mathsf{A} \cup \mathsf{R}$ where u is a prefix of v. By identical reasoning, there are no models where v is a prefix of u, and thus $\mathsf{S} \cup \mathsf{A} \cup \mathsf{R}$ is unsatisfiable in T_{SLRp}.

If the children of c are obtained by an application of D-Split, then each child is the root of a closed derivation tree. Since the lengths of u and v are entailed to be disequal, in all models of $\mathsf{S} \cup \mathsf{A} \cup \mathsf{R}$, either u is longer than v or vice versa. Using similar reasoning as for F-Split, by the induction hypothesis on the left branch, we have that there are no models of $\mathsf{S} \cup \mathsf{A} \cup \mathsf{R}$ where u is longer than v. Similarly, by the induction hypothesis on the right branch, we have there are no models where v is longer than u, and thus $\mathsf{S} \cup \mathsf{A} \cup \mathsf{R}$ is unsatisfiable in T_{SLRp}. □

We say a configuration C is saturated with respect to a rule R if either R does not apply to it, or all applications of R to C leave it unchanged modulo renaming of Skolem variables. A derivable configuration $\langle \mathsf{S}, \mathsf{A}, \mathsf{R}, \mathsf{F}, \mathsf{N}, \mathsf{C}, \mathsf{B} \rangle$ is *saturated* if (i) N is a total map over $\mathbf{E}_{\mathsf{S}}$, (ii) B is a partition of $\mathbf{E}_{\mathsf{S}}$, and (iii) it is saturated with respect to all rules other than Reset.

Theorem 2 (Solution soundness) *If a derivation tree with initial configuration $\langle S_0, A_0, \emptyset, \emptyset, \emptyset, \emptyset, \emptyset \rangle$ contains a saturated configuration then the set $S_0 \cup A_0$ is satisfiable in T_{SLRp}.*

Proof. Assume there exists a derivation tree with root node $\langle S_0, A_0, R_0, \emptyset, \emptyset, \emptyset, \emptyset \rangle$ containing a saturated configuration $\langle S_n, A_n, R_n, F_n, N_n, C_n, B_n \rangle$. We will show that we can build a model $\mathcal{M}$ of T_{SLRp} that satisfies $S_n \cup A_n \cup R_n$ (which is by construction a superset of $S_0 \cup A_0$). Since all models of T_{SLRp} interpret its function and predicate symbols in the same way, to build $\mathcal{M}$ we only need to define its interpretation of the variables in $S_n \cup A_n \cup R_n$.

Before assigning values to variables of type Str, we first construct a map $\mathcal{I}$ from terms of type Int in $S_n \cup A_n \cup R_n$ (that is, the variables of type Int in $\mathcal{V}(\mathsf{A})$, and terms $\mathsf{len}\, x$ for all $x \in \mathcal{V}(\mathsf{S})$) to integer values. Since A-Conflict does not apply to our configuration, we assume these values are chosen so that :

1. $(\neg)\mathcal{I}(t) \bowtie \mathcal{I}(s)$ for all $(\neg)t \bowtie s \in A_n$, where $\bowtie$ is one of $\approx, >$.
2. $\mathcal{I}(\mathsf{len}\, x) = \mathcal{I}(\mathsf{len}\, y)$ if and only if $\mathsf{len}\, x \approx \mathsf{len}\, y \in \mathcal{C}(S)_n$.

For all variables x of type Int in $\mathcal{V}(\mathsf{A})$, we say $\mathcal{M}(x) = \mathcal{I}(x)$.

Due to our selection of these assignments, and part 5 of Invariant 1, we have that $\mathcal{I}(len_{B_1}) \neq \mathcal{I}(len_{B_2})$ for all pairs of distinct buckets $B_1, B_2 \in B_n$. We sort all buckets in B_n to obtain the list $B_{i_1}, \ldots, B_{i_k}$ such that $\mathcal{I}(len_{B_{i_1}}) < \ldots < \mathcal{I}(len_{B_{i_k}})$. For $i = i_1, \ldots, i_k$ starting from i_1, we assign values to all variables occurring in the equivalence classes of B_i, which is described in the following. We will describe this process as incrementally assigning string constants to variables, and say a string constant l is *unused in* $\mathcal{M}$ if we have not assigned $\mathcal{M}(x) = l$ for any variable x. We maintain the invariant:

$$\mathcal{M}(\mathsf{len}\, x) = \mathcal{I}(\mathsf{len}\, x), \text{ for all variables } x \in \mathcal{V}(\mathsf{S}). \qquad (1)$$

First, consider equivalence classes $[x] \in B_i$ such that $N_n\,[x]$ is a tuple of the form $(a_1, \ldots, a_n)$, where $[x] \mapsto (a_1, \ldots, a_m)$ was added to N_n by an application of N-Form1, say for $F_n\, s = (a_1, \ldots, a_n)$ for some term $s \in [x]$. Since our configuration is saturated with respect to the rule Len, since $x \approx s \in \mathcal{C}(S)_n$, we have that $\mathcal{I}(a_{j_1}) + \ldots + \mathcal{I}(a_{j_m}) + k = \mathcal{I}(len_{B_i})$, where $\{a_{j_1}, \ldots, a_{j_m}\}$ are the variables in $\{a_1, \ldots, a_n\}$, and $k \geq 0$. Due our construction of $\mathcal{I}$, part 4(ii) of Invariant 1, and since our configuration is saturated with respect to Len-Split, we have that $\mathcal{I}(a_{j_1}) > 0 \wedge \ldots \wedge \mathcal{I}(a_{j_m}) > 0$. As a result, notice that $\mathcal{M}(\mathsf{con}(N_n\,[x]))$ (call this value l) is defined : if $m = 0$, then l is the empty string; otherwise, if $m > 1$, each of a_i is either a string constant or a variable such that $\mathcal{I}(\mathsf{len}\, a_i) < \mathcal{I}(len_{B_i})$, in which case a_i has been assigned a value in $\mathcal{M}$ due to the order in which buckets are processed. Thus, we say $\mathcal{M}(y) = l$ for each variable $y \in [x]$. By assumption of (1) for each of $a_{j_1}, \ldots, a_{j_m}$, we have that (1) is satisfied for y as well. Since our configuration is saturated with respect to the rule Card, either $|B_i| = 1$, or the value of $\mathcal{I}(len_{B_i})$ is at least $\lfloor \log_{|\mathcal{A}|}(|B_i| - 1) \rfloor + 1$. In either case, there exist at least $|B_i|$ string constants of length $\mathcal{I}(len_{B_i})$. Thus, for all other equivalence classes in $[x] \in B_i$ (the equivalence classes whose normal form is a variable in $[x]$), we may choose a string constant l that is unused in $\mathcal{M}$, and say $\mathcal{M}(y) = l$ for each variable $y \in [x]$, which clearly satisfies (1) as well.

We now argue why $\mathcal{M}$ satisfies $S_n \cup A_n \cup R_n$. Due to (1) and by construction of $\mathcal{M}$ for variables of sort Int in $\mathcal{V}(\mathsf{A})$, we have that $\mathcal{M}(s) = \mathcal{I}(s)$ for all terms $s \in \mathcal{D}(\mathcal{I})$. Thus, due to condition 1 of our construction of $\mathcal{I}$, we have that $\mathcal{M}$ satisfies A_n. Furthermore, since our configuration is saturated with respect to the rule S-Prop, $\mathcal{M}$ satisfies all equalities between terms of sort Int in S_n. Due to condition 2 of our construction of $\mathcal{I}$ and since S-Conflict does not apply, $\mathcal{M}$ satisfies all disequalities between terms of sort Int in S_n as well.

Since our configuration is saturated the rule R-Star, we have that $s \approx \epsilon \in \mathcal{C}(S)_n$ for all constraints of the form s in $\mathsf{star}(\mathsf{set}(t))$ in R_n. Moreover, all constraints in R_n (those introduced by applications of F-Loop) are of the form s in $\mathsf{star}(\mathsf{set}(t))$ for some strings s and t. Since ϵ in $\mathsf{star}(\mathsf{set}(t))$ is a tautology for any t, we have that $\mathcal{M}$ satisfies R_n.

To show that $\mathcal{M}$ satisfies the equalities between string terms in S_n, we begin with an intermediate lemma.

Lemma 3. $\mathcal{M}(t) = \mathcal{M}(N_n\,[t])$ *for all terms* t *of sort* Str *in* $\mathcal{T}(S_n)$.

Proof. We may assume the lemma holds for all s where $\mathcal{M}(\mathsf{len}\, s) < \mathcal{M}(\mathsf{len}\, t)$, and thus $\mathcal{M}$ satisfies all equalities in $\mathcal{C}(S)_n$ between terms whose lengths in $\mathcal{M}$ are strictly less than $\mathcal{M}(\mathsf{len}\, t)$. If t is a variable, the statement holds by our construction of $\mathcal{M}$. If t is not a variable, then by the premises of N-Form1 and N-Form2, either $F_n\, t = N_n\, t$, or $t \in C_n$. When $F_n\, t = N_n\, t$ where t is a string constant l, then $N_n\, t = (l){\downarrow}$, and the statement is immediate. When $F_n\, t = N_n\, t$ and t is a term of the form $\mathsf{con}(t_1, \ldots, t_n)$ for $n > 1$, then $F_n\, t = (\boldsymbol{s}_1, \ldots, \boldsymbol{s}_n)$, where $N_n\, t_1 = \boldsymbol{s}_1$, …, $N_n\, t_n = \boldsymbol{s}_n$. Since $n > 1$, our configuration is saturated with respect to the rule Len, and part 4(ii) of Invariant 1, we have that $\mathcal{M}(\mathsf{len}\, t_1) < \mathcal{M}(\mathsf{len}\, t)$, …, $\mathcal{M}(\mathsf{len}\, t_n) < \mathcal{M}(\mathsf{len}\, t)$. Thus, by our assumption $\mathcal{M}(t_1) = \mathcal{M}(\boldsymbol{s}_1)$, …, $\mathcal{M}(t_n) = \mathcal{M}(\boldsymbol{s}_n)$. Thus, $\mathcal{M}(t) = \mathcal{M}(\mathsf{con}(\mathcal{M}(\boldsymbol{s}_1), \ldots, \mathcal{M}(\boldsymbol{s}_n))) = \mathcal{M}(N_n\,[t])$. We have thus shown the lemma holds for each term $t \notin C_n$.

To show the lemma holds for each term $t \in C_n$, we first identify a *parent term* s distinct from t such that $t \approx s \in \mathcal{C}(S)_n$, and $\mathcal{M}(t) = \mathcal{M}(s)$. We will construct a (partial) function parent from terms to terms, where $\mathsf{parent}(t)$ denotes the parent term of t.

If $t \in C_n$, and t was added by an application of S-Cycle, then t is $\mathsf{con}(t_1, \ldots, t_i, \ldots, t_n)$ and since constraints are never removed from S, we have that $t_k \approx \epsilon \in \mathcal{C}(S)_n$ for each $k \in \{1, \ldots, n\} \setminus \{i\}$. We have that either $\mathcal{M}(\mathsf{len}\, t) = 0$, or by our assumption $\mathcal{M}$ satisfies each of $t_k \approx \epsilon$. In each of these cases, we have that $\mathcal{M}(t) = \mathcal{M}(t_i)$, and since $t \approx t_i \in \mathcal{C}(S)_n$, we say $\mathsf{parent}(t) = t_i$.

If $t \in C_n$, and t was added by an application of F-Loop, then due to part 4(ii) of Invariant 1, we have that $\mathcal{M}(\mathsf{len}\, x) < \mathcal{M}(\mathsf{len}\, t)$, $\mathcal{M}(\mathsf{len}\, \boldsymbol{u}_1) < \mathcal{M}(\mathsf{len}\, s) = \mathcal{M}(\mathsf{len}\, t)$, and $\mathcal{M}(\mathsf{len}\, \boldsymbol{v}_1) < \mathcal{M}(\mathsf{len}\, t)$. By our assumption, $\mathcal{M}(s) = \mathcal{M}(\mathsf{con}(\boldsymbol{w}, k_2, k, k_1, k_2, \boldsymbol{v}_2))$, and $\mathcal{M}(t) = \mathcal{M}(\mathsf{con}(\boldsymbol{w}, k_2, k_1, k_2, k, \boldsymbol{v}_2))$. Since $\mathcal{M}$ satisfies R_n, we have that $\mathcal{M}(k)$ is of the form $(\mathcal{M}(k_1)\mathcal{M}(k_2))^n$. For all such n, we have that $\mathcal{M}(t) = \mathcal{M}(s)$, and since $t \approx s \in \mathcal{C}(S)_n$, we say $\mathsf{parent}(t) = s$.

Consider the path from the initial configuration to our saturated configuration. We have that for all t where $\mathsf{parent}(t)$ is defined, $\mathsf{parent}(t)$ was added to S_n after t was. To see this, notice that S-Cycle removes $t_i = \mathsf{parent}(t)$ from C, and F-Loop guarantees that $s = \mathsf{parent}(t)$ is not in C since it has a flat form. Thus, we have that $\mathsf{parent}^n(t) \neq t$ for all terms t, $n > 0$, where $\mathsf{parent}^n(t)$ (the n-fold application of parent) is defined. For each $t \in C_n$, consider the largest n such that $\mathsf{parent}^n(t)$ is defined, where by the previous observation have that n is finite. We have that $\mathcal{M}(t) = \mathcal{M}(\mathsf{parent}(t)) = \ldots = \mathcal{M}(\mathsf{parent}^n(t))$, which since $\mathsf{parent}^n(t) \notin C_n$, is equal to $\mathcal{M}(N_n\,[\mathsf{parent}^n(t)])$. Additionally, we have that $t \approx \mathsf{parent}(t)$, …, $\mathsf{parent}^{n-1}(t) \approx \mathsf{parent}^n(t) \in \mathcal{C}(S)_n$, and thus $[t] = [\mathsf{parent}^n(t)]$. Putting these together, we have $\mathcal{M}(t) = \mathcal{M}(N_n\,[t])$. ☐

Due to this lemma, and since N_n is a total map, $\mathcal{M}$ satisfies all equalities between terms of type Str in S_n. To show $\mathcal{M}$ also satisfies the disequalities between string terms in S_n, we show the following lemma.

Lemma 4. $\mathcal{M}(\mathsf{con}(N_n\,[x])) \neq \mathcal{M}(\mathsf{con}(N_n\,[y]))$ *for each pair of distinct equivalence classes* $[x], [y]$ *in any bucket* B.

Proof. We may assume that the lemma holds for all buckets B' where $\mathcal{M}(len_{B'}) < \mathcal{M}(len_B)$. Since $[x]$ and $[y]$ occur in the same bucket, due to our premises in the rule D-Add, $N_n\,[x]$ and $N_n\,[y]$ must be of the form $(\boldsymbol{w}, u, \boldsymbol{u}_1)$ and $(\boldsymbol{w}, v, \boldsymbol{v}_1)$ respectively, where $S_n \models \mathsf{len}\,u \approx \mathsf{len}\,v$ and $u \not\approx v \in \mathcal{C}(S)_n$. If $\boldsymbol{w}, \boldsymbol{u}_1, \boldsymbol{v}_1$ are empty tuples, then each of u and v is either a variable or a string constant, and the lemma holds by our construction of $\mathcal{M}$. Otherwise, we have that $\mathcal{M}(\mathsf{len}\,u) < \mathcal{M}(len_B)$ and $\mathcal{M}(\mathsf{len}\,v) < \mathcal{M}(len_B)$, and moreover since u and v have the same length, then $[u]$ and $[v]$ occur in the same bucket B'. Due to our assumption for bucket B', we have that $\mathcal{M}(\mathsf{con}(N_n\,[u])) \neq \mathcal{M}(\mathsf{con}(N_n\,[v]))$ and thus $\mathcal{M}(\mathsf{con}(N_n\,[x])) \neq \mathcal{M}(\mathsf{con}(N_n\,[y]))$ as well. ☐

Due to part 5 of Invariant 1 and by our construction of $\mathcal{M}$, we have that $\mathcal{M}(\mathsf{len}\,N_n\,[x]) = \mathcal{M}(\mathsf{len}\,N_n\,[y])$ if and only if $[x]$ and $[y]$ occur in the same bucket. Thus, by the previous two lemmas, we have that $\mathcal{M}(s) \neq \mathcal{M}(t)$ for all pair of terms s, t where s and t reside in distinct equivalence classes of $\mathcal{C}(S)_n$. Since S-Conflict does not apply, we have that $\mathcal{M}$ satisfies all the disequalities between terms of type Str in S_n as well. This concludes the proof of Theorem 2. ☐

Proof procedure A possible proof procedure, a highly simplified version of the one we have implemented, is defined by the repeated application of the calculus rules according to the six steps below. When applying a branching rule the procedure tries the left-branch configuration first. It interrupts a step and restarts with Step 0 as soon as a constraint is added to S. The procedure keeps cycling through the steps until it derives a saturated configuration or the unsat one. In the latter case, it continues with another configuration in the derivation tree, if any.

Step 0: Reset: Apply Reset to reset buckets, and flat and normal forms.

Step 1: Check for conflicts, propagate: Apply S-Conflict or A-Conflict if the configuration is unsatisfiable due to the current string or arithmetic constraints; otherwise, propagate entailed equalities between S and A using S-Prop and A-Prop.

Step 2: Add length constraints: Apply Len and then Len-Split to completion.

Step 3: Compute Normal Forms for Equivalence Classes. Apply S-Cycle to completion and then the rules in Figure 4 to completion. If this does not produce a total map N, there must be some $s \approx t \in \mathcal{C}(\mathsf{S})$ such that $\mathsf{F}\,s$ and $\mathsf{F}\,t$ have respectively the form

$(\boldsymbol{w}, u, \boldsymbol{u}_1)$ and $(\boldsymbol{w}, v, \boldsymbol{v}_1)$ with u and v distinct terms. Let x, y be variables with $x \in [u]$ and $y \in [v]$. If S entails neither $\mathsf{len}\, x \approx \mathsf{len}\, y$ nor $\mathsf{len}\, x \approx \mathsf{len}\, y$, apply L-Split to them; otherwise, apply any applicable rules from Figure 5, giving preference to F-Unify.

Step 4: Partition equivalence classes into buckets. First apply D-Base and D-Add to completion. If this does not make B a partition of $\mathbf{E}_{\mathsf{S}}$, there must be an equivalence class $[x]$ contained in no bucket but such that $S \models \mathsf{len}\, x \approx len_B$ for some bucket B (otherwise D-Base would apply). If there is a $[y] \in B$ such that $x \not\approx y \notin \mathcal{C}(\mathsf{S})$, split on $x \approx y$ and $x \not\approx y$ using S-Split. Otherwise, let $[y] \in B$ such that $x \not\approx y \in \mathcal{C}(\mathsf{S})$. It must be that $\mathsf{N}\,[x]$ and $\mathsf{N}\,[y]$ share a prefix followed by two distinct terms u and v. Let x_u, x_v be variables with $x_u \in [u]$ and $x_v \in [v]$. If $S \models \mathsf{len}\, x_u \not\approx \mathsf{len}\, x_v$, apply the rule D-Split to u and v. If $S \models \mathsf{len}\, x_u \approx \mathsf{len}\, x_v$, since it is also the case that neither $x_u \approx x_v$ nor $x_u \not\approx x_v$ is in $\mathcal{C}(\mathsf{S})$, apply S-Split to x_u and x_v. If S entails neither $\mathsf{len}\, x_u \approx \mathsf{len}\, x_v$ nor $\mathsf{len}\, x_u \not\approx \mathsf{len}\, x_v$, split on them using L-Split.

Step 5: Add length constraint for cardinality. Apply Card to completion.

One can show that all derivation trees generated with this proof procedure satisfy Invariant 1. We illustrate the procedure's workings with a couple of examples.

Example 1. Suppose we start with $\mathsf{A} = \emptyset$ and $\mathsf{S} = \{\mathsf{len}\, x \approx \mathsf{len}\, y,\ x \not\approx \epsilon,\ z \not\approx \epsilon,\ \mathsf{con}(x, l_1, z) \approx \mathsf{con}(y, l_2, z)\}$ where l_1, l_2 are distinct constants of the same length. After checking for conflicts, the procedure applies Len and Len-Split to completion. All resulting derivation tree branches except one can be closed with S-Conflict. In the leaf of the non-closed branch every string variable is in a disequality with ϵ. In that configuration, the string equivalence classes are $\{x\}$, $\{y\}$, $\{z\}$, $\{l_1\}$, $\{l_2\}$, $\{\epsilon\}$, and $\{\mathsf{con}(x, l_1, z), \mathsf{con}(y, l_2, z)\}$. The normal form for the first three classes is computed with N-Form2; the normal form for the other three with F-Form2 and N-Form1. For the last equivalence class, the procedure uses F-Form1 to construct the flat forms $\mathsf{F}\, \mathsf{con}(x, l_1, z) = (x, l_1, z)$ and $\mathsf{F}\, \mathsf{con}(y, l_2, z) = (y, l_2, z)$, and F-Unify to add the equality $x \approx y$ to S. The procedure then restarts but now with the string equivalence classes $\{x, y\}$, $\{z\}$, $\{l_1\}$, $\{l_2\}$, $\{\epsilon\}$, and $\{\mathsf{con}(x, l_1, z), \mathsf{con}(y, l_2, z)\}$. After similar steps as before, the terms in the last equivalence class get the flat form (x, l_1, z) and (x, l_2, z) respectively (assuming x is chosen as the representative term for $\{x, y\}$). Using F-Unify, the procedure adds the equality $l_1 \approx l_2$ to S and then derives unsat with S-Conflict. This closes the derivation tree, showing that the input constraints are unsatisfiable. □

Example 2. Suppose now the input constraints are $\mathsf{A} = \emptyset$ and $\mathsf{S} = \{\mathsf{len}\, x \approx \mathsf{len}\, y,\ x \not\approx \epsilon,\ z \not\approx \epsilon,\ \mathsf{con}(x, l_1, z) \not\approx \mathsf{con}(y, l_2, z)\}$ with l_1, l_2 as in Example 1. After similar steps as in that example, the procedure can derive a configuration where the string equivalence classes are $\{x\}$, $\{y\}$, $\{z\}$, $\{l_1\}$, $\{l_2\}$, $\{\epsilon\}$, $\{\mathsf{con}(x, l_1, z)\}$, and $\{\mathsf{con}(y, l_2, z)\}$. After computing normal forms for these classes, it attempts to construct a partition B of them into buckets. However, notice that if it adds $\{[x]\}$, say, to B using D-Base, then neither D-Base (since $S \models \mathsf{len}\, x \approx \mathsf{len}\, y$) nor D-Add (since $x \not\approx y \notin \mathcal{C}(\mathsf{S})$) is applicable to $[y]$. So it applies S-Split to x and y. In the branch where $x \approx y$, the proof procedure subsequently restarts, and computes normal forms as before. At that point it succeeds in making B a partition of the string equivalence classes, by placing $[\mathsf{con}(x, l_1, z)]$ and $[\mathsf{con}(y, l_2, z)]$ into the same bucket using D-Add, which applies because their corresponding normal forms are (x, l_1, z) and (x, l_2, z) respectively. Any further rule applications lead to branches with a saturated configuration, each of which indicates that the input constraints are satisfiable. □

Implementation in DPLL(T) Theory solvers based on the calculus we have described can be integrated into the DPLL(T) framework used by modern SMT solvers, which combines a SAT solver with multiple specialized *theory solvers* for conjunctions of constraints in a certain theory. These SMT solvers maintain an evolving set F of quantifier-free clauses and a set M of literals representing a (partial) Boolean assignment for F. Periodically, a theory solver is asked whether M is satisfiable in its theory.

In terms of our calculus, we assume that the literals of an assignment M are partitioned into string constraints (corresponding to the set S), arithmetic constraints (the set A) and RL constraints (the set R). These sets are subsequently given to three independent solvers, which we will call the string solver, the arithmetic solver, and the RL solver, respectively. The rules A-Prop and S-Prop model the standard mechanism for Nelson-Oppen theory combination, where entailed equalities are communicated between these solvers. The satisfiability check performed by the arithmetic solver is modeled by the rule A-Conflict. Note that there is no additional requirement on the arithmetic solver, and thus a standard DPLL(T) theory solver for linear integer arithmetic can be used. The behavior of the RL solver is described by the rule R-Star and others we have omitted here. The remaining rules model the behavior of the string solver.

The case splitting done by the string solver (with rules S-Split and L-Split) is achieved by means of the *splitting on demand* paradigm [1], in which a solver may add theory lemmas to F consisting of clauses possibly with literals not occurring in M. The case splitting in rules F-Split and D-Split can be implemented by adding a lemma of the form $\psi \Rightarrow (l_1 \vee l_2)$ to F, where l_1 and l_2 are new literals. For instance, in the case of F-Split, we add the lemma $\psi \Rightarrow (u \approx \mathsf{con}(v, z) \vee v \approx \mathsf{con}(u, z))$, where ψ is a conjunction of literals in M entailing $s \approx t \wedge s \approx \mathsf{F}\, s \wedge t \approx \mathsf{F}\, t \wedge \mathsf{len}\, u \not\approx \mathsf{len}\, v$ in the overall theory.

The rules Len, Len-Split, and Card involve adding constraints to A. This is done by the string solver by adding lemmas to F containing arithmetic constraints. For instance, if $x \approx \mathsf{con}(y, z) \in \mathcal{C}(\mathsf{S})$, the solver may add a lemma of the form $\psi \Rightarrow \mathsf{len}\, x \approx \mathsf{len}\, y + \mathsf{len}\, z$ to F, where ψ is a conjunction of literals from M entailing $x \approx \mathsf{con}(y, z)$, after which the conclusion of this lemma is added to M (and hence to A).

In DPLL(T), when a theory solver determines that M is unsatisfiable (in the solver's theory) it generates a *conflict clause*, the negation of an unsatisfiable subset of M. The string solver maintains a compact representation of $\mathcal{C}(\mathsf{S})$ at all times. To construct conflict clauses it also maintains an *explanation* $\psi_{s,t}$ for each equality $s \approx t$ it adds to S by applying S-Cycle, F-Unify or standard congruence closure rules. The explanation $\psi_{s,t}$ is a conjunction of string constraints in M such that $\psi_{s,t} \models_{\mathsf{SLRp}} s \approx t$. For F-Unify, the string solver maintains an explanation ψ for the flat form of each term $t \in \mathcal{D}(\mathsf{F})$ where $\psi \models_{\mathsf{SLRp}} t \approx \mathsf{con}(\mathsf{F}\, t)$. When a configuration is determined to be unsatisfiable by S-Conflict, that is, when $s \approx t, s \not\approx t \in \mathcal{C}(\mathsf{S})$ for some s, t, it replaces the occurrence of $s \approx t$ with its corresponding explanation ψ, and then replaces the equalities in ψ with their corresponding explanation, and so on, until ψ contains only equalities from M. Then it reports as a conflict clause (the clause form of) $\psi \Rightarrow s \approx t$.

All other rules (such as those that modify N, F and B) model the internal behavior of the string solver.

3 Experimental Results

We have implemented a theory solver based on the calculus and proof procedure described in the previous section within the latest version of our SMT solver CVC4. The string alphabet $\mathcal{A}$ for this implementation is the set of all 256 ASCII characters. To evaluate our solver we did an experimental comparison with two of the string solvers mentioned in Section 1.1: Z3-STR (version 20140120) and Kaluza (latest version from its website). These solvers, which have been widely used in security analysis, were chosen because they are publicly available and have an input language that largely intersects with that of our solver. All results in this section were collected on a 2.53 GHz Intel Xeon E5540 with 8 MB cache and 12 GB main memory.[5]

Modulo superficial differences in the concrete input syntax, all three tools accept as input a set of T_{SLRp} constraints and report on its satisfiability with a sat, unsat or unknown answer. In the first case, CVC4 and Z3-STR can also provide a *solution*, i.e., a satisfying assignment for the variables in the input set. Kaluza can do that for at most one *query variable* which must be specified before-hand in the input file.

An initial series of regression tests on all three tools revealed several usability and correctness issues with Kaluza and a few with Z3-STR. In Kaluza, they were caused by bugs in its top level script which communicates with different tools, e.g. the solvers Yices and Hampi, via the file system. They range from failure to clean up temporary files to an incorrect use of the Unix grep tool to extract information from the output of those tools. Since Kaluza is not in active development anymore, we made an earnest, best effort attempt to fix these bugs ourselves. However, there seem to be more serious flaws in Kaluza's interface or algorithm. Specifically, often Kaluza incorrectly reports unsat for problems that are satisfiable only if some of their input variables are assigned the empty string. Moreover, in several cases, Kaluza's sat/unsat answer for the same input problem changes depending on the query variable chosen. Because of this arbitrariness, in our experiments we removed all query variables in Kaluza's input.

We found that in several cases Z3-STR returns *spurious solutions*, assignments to the input variables that do not in fact satisfy the input problem. Also, it classifies some satisfiable problems as unsat. Prompted by our inquiries, the Z3-STR developers have produced a new version of Z3-STR that fixes the spurious solutions problem. Unfortunately, that version was not ready in time for us to redo the experiments. As for Z3-STR's unsoundness, it looks like it is caused by an internal restriction that, for efficiency but without loss of generality, limits the possible values of "free" string variables to a fixed finite set of string constants. The authors define a variable as free in an input problem if its values are completely unconstrained by the problem. For instance, in the constraint set $\{x \approx \mathsf{con}(y, z)\}$ variables y and z would be free according to this definition, while x would not. It appears that the criterion used by Z3-STR to recognize free variables sometimes misclassifies a variable as free when in fact it is not, causing the system to miss solutions that are outside the finite domain imposed on free variables.

In contrast, on our full set of benchmarks, we did not find any evidence of erroneous behavior in CVC4 when compared with the other two solvers. Every solution produced

[5] Detailed results and binaries can be found at http://cvc4.cs.nyu.edu/papers/CAV2014-strings/.

	CVC4	Z3-str		Kaluza		Kaluza-orig	
Result		×	✓	×	✓	×	✓
unsat	11,625	317	11,769	7,154	13,435	27,450	805
sat	33,271	1,583	31,372	n/a	25,468	n/a	3
unknown	0		0		3		0
timeout	2,388		2,123		84		84
error	0		120		1,140		18,942

Table 1. Comparative results.

by CVC4 was *confirmed* by both CVC4 and Z3-STR by adding the solution as a set of constraints to the input problem and checking that the strengthened problem was satisfiable. Furthermore, no unsat answers from CVC4 were contradicted by a confirmed solution from Z3-STR.

Comparative Evaluation For our evaluation we selected 47,284 benchmark problems from a set of about 50K benchmarks generated by Kudzu, a symbolic execution framework for Javascript, and available on the Kaluza website [**?**]. The discarded problems either had syntax errors or included a macro function (CapturedBrack) whose meaning is not fully documented. We translated those benchmarks into CVC4's extension of the SMT-LIB 2 format to the language of T_{SLRp}[6] and into the Z3-STR format. Some benchmarks contain regular membership constraints $(s \text{ in } r)$, which Z3-STR does not support. However, in all of these constraints the regular language denoted by r is finite and small, so we were able to translate them into equivalent string constraints.

We ran CVC4, Z3-STR and two versions of Kaluza, the original one and the one with our debugged main script, on each benchmark with a 20-second CPU time limit. The results are summarized in Table 1. There, the column Kaluza-orig refers to the original version of Kaluza while the error line counts the total number of runtime errors. The results for Z3-STR and the two versions of Kaluza are separated in two columns: the × column contains the number of provably incorrect answers while the ✓ column contains the rest. By *provably incorrect* here we mean an unsat answer for a problem that has a verified solution or a sat answer but with a spurious solution. Note that the figures for the two versions of Kaluza are unfairly skewed in their favor because neither version returns solutions, which means that their sat answers are unverifiable unless one of the other solvers produces a solution for the same problem. For a more detailed discussion, we look at the benchmark problem set broken down by the CVC4 results. For brevity we discuss only our amended version of Kaluza below.

None of the 11,625 unsat answers provided by CVC4 were provably incorrect. Z3-STR also answered sat on 11,568 of them and returned an error for the remaining 57; Kaluza agreed on 11,394 and returned an error for the rest. All of CVC4's 33,271 sat answers were corroborated by a confirmed solution. Z3-STR agreed on 31,616 of those problems although it returned a spurious solution for 244 of them. Also, it incorrectly

[6] The SMT-LIB 2 standard does not include a theory of strings yet although there are plans to do so. CVC4's extension is documented at `http://cvc4.cs.nyu.edu/wiki/Strings`.

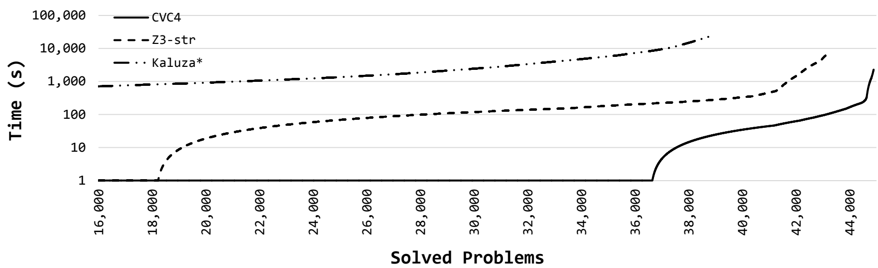

Fig. 7. Runtime comparison of CVC4, Z3-STR and the amended Kaluza. Times are in seconds.

found 317 problems unsatisfiable and produced an error on 29 problems, timing out on the remaining 1,304. Kaluza agreed on 25,468 problems (unverifiable because of the absence of solutions), erroneously classified 7,154 as unsatisfiable, reported unknown for 3, produced an error for 562, and timed out on 84.

CVC4 timed out on 2,388 problems, but produced no errors and no unknown answers. For the problems that CVC4 timed out on, Z3-STR classified 201 as unsatisfiable, returned an error for 34 and produced solutions for the remaining 1,339, all of which were spurious. Kaluza classified 2,041 as unsatisfiable and returned an error on the rest.

These results provide strong evidence that CVC4's string solver is sound. They also provide evidence that unsat answers from Z3-STR and Kaluza for problems on which CVC4 times out cannot be trusted. They also show that CVC4's string solver answers sat more often than both Z3-STR and Kaluza, providing a correct solution in each case. Thus, it is overall the best tool for both satisfiable and unsatisfiable problems.

Moving to run time performance, a comparison with Kaluza is not very meaningful because of its high unreliability and the unverifiability of its sat answers. In principle, the same could be said of Z3-STR due to its refutation unsoundness.[7] However, an analysis of our detailed results shows that CVC4 has nonetheless better runtime performance overall. This can be easily seen from the cactus plot in Figure 7, which shows for each of the three systems how many non-provably incorrect benchmarks it cumulatively solves within a certain amount of time.

4 Conclusion and Further Work

We have presented a new approach for solving quantifier-free constraints over a theory of unbounded strings with length and regular language membership. Our approach integrates a specialized theory solver for such constraints within the DPLL(T) framework. We have given experimental evidence that our implementation in the SMT solver CVC4 is highly competitive with existing tools.

In our ongoing work, we plan to extend the scope of our string solver to support a richer language of string constraints that occur often in practice, especially in security

[7] Z3-STR could be faster and time out less often simply because it unduly prunes search space.

applications. In preliminary implementation work in CVC4, we have found that commonly used predicates (such as the predicate contains for string containment) can be handled in an efficient manner by extending the calculus mentioned in this paper. We are also working on a more sophisticated approach for dealing with RL constraints, using a separate dedicated solver that is similarly integrated into the DPLL(T) framework.

At the theoretical level, we would like to devise a proof strategy that is solution-complete, that is, guaranteed to eventually produce a solution for every satisfiable input. Note that a fair proof strategy can be trivially obtained by incrementally setting an upper bound on the total length of all strings in a problem solution. The challenge is to devise a more efficient fair strategy than that one. Additionally, we would like to identify fragments where our calculus is terminating, and thus refutation complete.

Acknowledgments We would like to thank Nestan Tsiskaridze for her insightful comments, and the developers of Z3-STR for their technical support in using their tool and several clarifications on it.

References

[1] C. Barrett, R. Nieuwenhuis, A. Oliveras, and C. Tinelli. Splitting on demand in SAT modulo theories. In *Proceedings of LPAR'06*, volume 4246 of *LNCS*, pages 512–526. Springer, 2006.

[2] C. Barrett, R. Sebastiani, S. Seshia, and C. Tinelli. Satisfiability modulo theories. In A. Biere, M. J. H. Heule, H. van Maaren, and T. Walsh, editors, *Handbook of Satisfiability*, volume 185, chapter 26, pages 825–885. IOS Press, February 2009.

[3] N. Bjørner, N. Tillmann, and A. Voronkov. Path feasibility analysis for string-manipulating programs. In *Proceedings of the 15th International Conference on Tools and Algorithms for the Construction and Analysis of Systems: Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009,*, pages 307–321. Springer-Verlag, 2009.

[4] A. S. Christensen, A. Møller, and M. I. Schwartzbach. Precise analysis of string expressions. In *Proceedings of the 10th International Conference on Static Analysis*, SAS'03, pages 1–18, Berlin, Heidelberg, 2003. Springer-Verlag.

[5] X. Fu and C. Li. A string constraint solver for detecting web application vulnerability. In *Proceedings of the 22nd International Conference on Software Engineering and Knowledge Engineering*, SEKE'2010. Knowledge Systems Institute Graduate School, 2010.

[6] V. Ganesh, M. Minnes, A. Solar-Lezama, and M. Rinard. Word equations with length constraints: What's decidable? In *Proceedings of the 8th International Conference on Hardware and Software: Verification and Testing*, HVC'12, pages 209–226, Berlin, Heidelberg, 2013. Springer-Verlag.

[7] I. Ghosh, N. Shafiei, G. Li, and W. Chiang. JST: An automatic test generation tool for industrial Java applications with strings. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 992–1001, Piscataway, NJ, USA, 2013. IEEE Press.

[8] P. Hooimeijer and M. Veanes. An evaluation of automata algorithms for string analysis. In *Proceedings of the 12th international conference on Verification, model checking, and abstract interpretation*, pages 248–262. Springer-Verlag, 2011.

[9] P. Hooimeijer and W. Weimer. A decision procedure for subset constraints over regular languages. In *Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*, pages 188–198. ACM, 2009.

[10] P. Hooimeijer and W. Weimer. Solving string constraints lazily. In *Proceedings of the IEEE/ACM international conference on Automated software engineering*, pages 377–386. ACM, 2010.

[11] 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*, pages 105–116. ACM, 2009.

[12] G. Li and I. Ghosh. PASS: String solving with parameterized array and interval automaton. In V. Bertacco and A. Legay, editors, *Hardware and Software: Verification and Testing*, volume 8244 of *Lecture Notes in Computer Science*, pages 15–31. Springer International Publishing, 2013.

[13] G. S. Makanin. The problem of solvability of equations in a free semigroup. *English transl. in Math USSR Sbornik*, 32:147–236, 1977.

[14] G. Nelson and D. C. Oppen. Simplification by cooperating decision procedures. *ACM Trans. on Programming Languages and Systems*, 1(2):245–257, Oct. 1979.

[15] R. Nieuwenhuis, A. Oliveras, and C. Tinelli. Solving SAT and SAT Modulo Theories: from an abstract Davis-Putnam-Logemann-Loveland Procedure to DPLL(T). *Journal of the ACM*, 53(6):937–977, Nov. 2006.

[16] W. Plandowski. Satisfiability of word equations with constants is in pspace. *J. ACM*, 51(3):483–496, May 2004.

[17] P. Saxena, D. Akhawe, S. Hanna, F. Mao, S. McCamant, and D. Song. A symbolic execution framework for JavaScript. In *Proceedings of the 2010 IEEE Symposium on Security and Privacy*, pages 513–528. IEEE Computer Society, 2010.

[18] N. Tillmann and J. Halleux. Pex - white box test generation for .net. In B. Beckert and R. Hähnle, editors, *Tests and Proofs*, volume 4966 of *Lecture Notes in Computer Science*, pages 134–153. Springer Berlin Heidelberg, 2008.

[19] M. Veanes. Applications of symbolic finite automata. In *Proceedings of the 18th International Conference on Implementation and Application of Automata*, CIAA'13, pages 16–23, Berlin, Heidelberg, 2013. Springer-Verlag.

[20] M. Veanes, N. Bjørner, and L. De Moura. Symbolic automata constraint solving. In *Proceedings of the 17th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning*, LPAR'10, pages 640–654, Berlin, Heidelberg, 2010. Springer-Verlag.

[21] F. Yu, M. Alkhalaf, and T. Bultan. Stranger: An automata-based string analysis tool for php. In J. Esparza and R. Majumdar, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 6015 of *Lecture Notes in Computer Science*, pages 154–157. Springer Berlin Heidelberg, 2010.

[22] Y. Zheng, X. Zhang, and V. Ganesh. Z3-str: A z3-based string solver for web application analysis. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, pages 114–124, New York, NY, USA, 2013. ACM.

A F-Loop rule

Lemma 5. *For all string constants* l, l_1, l_2 *such that* $\mathsf{con}(l, l_2) = \mathsf{con}(l_1, l)$, $l_1 \neq \epsilon$, *and* $l_2 \neq \epsilon$, *there exists string constants* k, k_1, k_2 *such that* $l_1 = \mathsf{con}(k_2, k_1)$, $l_2 = \mathsf{con}(k_1, k_2)$, $l = \mathsf{con}(k_2, k)$, *and* $k \in \mathsf{star}(\mathsf{set}(\mathsf{con}(k_1, k_2)))$.

Proof. Assume without loss of generality that $l_1 = \mathsf{con}(k_2, k_1)$ for some constants k_1, k_2 where $\mathsf{len}\, k_2 = \mathsf{len}\, l \bmod \mathsf{len}\, l_1$. This means $\mathsf{len}\, k_2 = \mathsf{len}\, l - n * \mathsf{len}\, l_1$ for some integer $n \geq 0$, and thus we have $\mathsf{len}\, l = n * \mathsf{len}\, l_1 + \mathsf{len}\, k_2$. We prove the lemma holds for all l such that $\mathsf{con}(l, l_2) = \mathsf{con}(k_2, k_1, l)$, by induction on n.

If $n = 0$, we have that $\mathsf{len}\, l = \mathsf{len}\, k_2$, and thus $l = \mathsf{con}(k_2, \epsilon)$, $l_2 = \mathsf{con}(k_1, k_2)$, and $\epsilon \in \mathsf{star}(\mathsf{set}(\mathsf{con}(k_1, k_2)))$. For $n > 0$, let $l = \mathsf{con}(l', l'')$, where $\mathsf{len}\, l' = \mathsf{len}\, l_1$ and $\mathsf{len}\, l'' = (n-1) * \mathsf{len}\, l_1 + \mathsf{len}\, k_2$. By expanding our assumption, we have that $\mathsf{con}(l', l'', l_2) = \mathsf{con}(k_2, k_1, l', l'')$. Since $\mathsf{len}\, l' = \mathsf{len}\, l_1$, we have that $l' = \mathsf{con}(k_2, k_1)$, and $\mathsf{con}(l'', l_2) = \mathsf{con}(k_2, k_1, l'')$. Since $\mathsf{len}\, l'' = (n-1) * \mathsf{len}\, l_1 + \mathsf{len}\, k_2$, by the induction hypothesis we have that $l_2 = \mathsf{con}(k_1, k_2)$, $l'' = \mathsf{con}(k_2, k)$, and $k \in \mathsf{star}(\mathsf{set}(\mathsf{con}(k_1, k_2)))$ for some k. Due to the length of l'', we have that $l'' = \mathsf{con}(k_2, \mathsf{con}(k_1, k_2)^{(n-1)})$. Thus, we have $l = \mathsf{con}(k_2, k_1, k_2, \mathsf{con}(k_1, k_2)^{(n-1)}) = \mathsf{con}(k_2, k')$, where $k' = \mathsf{con}(k_1, k_2)^n$ $\in \mathsf{star}(\mathsf{set}(\mathsf{con}(k_1, k_2)))$. □

Corollary 1 *If* $\langle \mathsf{S}', \mathsf{A}, \mathsf{R}', \mathsf{F}, \mathsf{N}, \mathsf{C}', \mathsf{B} \rangle$ *is the result of applying* F-Loop *to the configuration* $\langle \mathsf{S}, \mathsf{A}, \mathsf{R}, \mathsf{F}, \mathsf{N}, \mathsf{C}, \mathsf{B} \rangle$*, then* $\mathsf{S}' \cup \mathsf{A} \cup \mathsf{R}'$ *is equisatisfiable modulo* T_{SLRp} *to* $\mathsf{S} \cup \mathsf{A} \cup \mathsf{R}$.

Proof. Since $s \approx t \in \mathcal{C}(\mathsf{S})$ and due to part 4 of Invariant 1, we have that $\mathsf{S} \models_{\mathsf{SLRp}} \mathsf{con}(x, \boldsymbol{u}_1) \approx \mathsf{con}(v, \boldsymbol{v}_1, x, \boldsymbol{v}_2)$. Thus, $\mathsf{S} \models_{\mathsf{SLRp}} \mathsf{con}(\boldsymbol{u}_1) \approx \mathsf{con}(\boldsymbol{u}_1', \boldsymbol{v}_2)$, and $\mathsf{S} \models_{\mathsf{SLRp}} \mathsf{con}(x, \boldsymbol{u}_1') \approx \mathsf{con}(v, \boldsymbol{v}_1, x)$ for some $\boldsymbol{u}_1'$. Due to part 4 of Invariant 1, we have that $\mathsf{S} \models_{\mathsf{SLRp}} \mathsf{con}(v, \boldsymbol{v}_1) \not\approx \epsilon$. Due to the previous lemma, we have that all models of S satisfy $\mathsf{con}(v, \boldsymbol{v}_1) \approx \mathsf{con}(k_2, k_1)$, $\mathsf{con}(\boldsymbol{u}_1') \approx \mathsf{con}(k_1, k_2)$, $x \approx \mathsf{con}(k_2, k)$, and $k \in \mathsf{star}(\mathsf{con}(\mathsf{set}\, k_1, \mathsf{set}\, k_2))$ for some k, k_1, k_2. Thus, we have that S entails $\mathsf{con}(v, \boldsymbol{v}_1) \approx \mathsf{con}(k_2, k_1)$, $\mathsf{con}(\boldsymbol{u}_1) \approx \mathsf{con}(k_1, k_2, \boldsymbol{v}_2)$, $x \approx \mathsf{con}(k_2, k)$, and $k \in \mathsf{star}(\mathsf{con}(\mathsf{set}\, k_1, \mathsf{set}\, k_2))$ for fresh variables k, k_1, k_2, or in other words, $(\mathsf{S}' \cup \mathsf{A} \cup \mathsf{R}') \setminus (\mathsf{S} \cup \mathsf{A} \cup \mathsf{R})$. Thus, the corollary holds. □