Form Methods Syst Des
DOI 10.1007/s10703-016-0247-6

An efficient SMT solver for string constraints

Tianyi Liang[1,2] · **Andrew Reynolds**[1] · **Nestan Tsiskaridze**[1] · **Cesare Tinelli**[1] · **Clark Barrett**[3] · **Morgan Deters**[3]

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 diverse set of data types that includes character strings. Until recently, satisfiability solvers for strings were standalone tools that could reason only about fairly restricted fragments of the theory of strings and regular expressions (e.g., strings of bounded lengths). These solvers were 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 a rich 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 common DPLL(T) architecture. We have implemented them in our SMT solver CVC4, expanding its already large set of built-in theories to include a theory of strings with concatenation, length, and membership in regular languages. This implementation makes CVC4 the first solver able to accept a rich set of mixed constraints over strings, integers, reals, arrays and algebraic datatypes. Our initial experimental results show that, in addition, on

This paper is dedicated to the memory of Morgan Deters who died unexpectedly in January 2015.

✉ Cesare Tinelli
cesare-tinelli@uiowa.edu

Tianyi Liang
liangtianyi@gmail.com

Andrew Reynolds
andrew-reynolds@uiowa.edu

Nestan Tsiskaridze
nestan-tsiskaridze@uiowa.edu

Clark Barrett
barrett@cs.nyu.edu

1 Department of Computer Science, The University of Iowa, Iowa, USA

2 Present Address: Two Sigma Investments, New York, USA

3 Department of Computer Science, New York University, New York, USA

Published online: 05 April 2016

pure string problems CVC4 is highly competitive with specialized string solvers accepting a comparable input language.

Keywords String solving · Satisfiability modulo theories · Automated deduction

Mathematics Subject Classification 68Q60 · 03B10 · 03B20 · 03B22 · 03B25 · 03B70

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 [5,6,9,21]. These techniques have benefited from the rise of powerful specialized reasoning engines such as SMT solvers. Security analyses frequently require the ability 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 can benefit from solvers able to 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, 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 [4, 24]. 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 because their possible models can be finitely enumerated, but also some fragments over unbounded strings (e.g., word equations [20]). Recent research has focused on identifying decidable fragments suitable for program analysis and, more crucially, on developing efficient solvers for them. Unfortunately, until very recently the available string solvers were standalone tools that could reason only about (some fragment of) the theory of strings and regular expressions. Often, they imposed strong restrictions on the expressiveness of their input language such as, for instance, exact length bounds on all string variables. Traditionally, these solvers were based on reductions to satisfiability problems over other data types, such as bit vectors, or to decision problems over automata.

In the last couple of years, alternative approaches have been developed that are based on algebraic techniques for solving (quantifier-free) constraints natively over a theory of unbounded strings with length and regular language membership [1,18,35]. We present one such approach in this paper. One of its distinguishing features is that it is geared towards the construction of string solvers that can be integrated into general, multi-theory SMT solvers based on the common DPLL(T) architecture [23]. We have built a solver based on our approach within the DPLL(T)-based SMT solver CVC4. To our knowledge, CVC4 is the first solver able to reason about a language of mixed constraints that includes strings together with integers, reals, bit vectors, arrays, and algebraic datatypes. The experimental results we present in this paper show that, in addition, over pure string problems the performance and reliability of CVC4 is superior to that of specialized string solvers able to reason about the same fragment of the theory of strings.

We describe our approach abstractly in terms of derivation rules and a proof procedure for the calculus consisting of these rules. After discussing related work, we define in Sect. 2 the theory of strings and regular expressions we work with, presenting a calculus for this theory in Sect. 3. We prove a number of correctness results about the calculus in Sect. 4 and provide a general proof procedure for it in Sect. 5. Our string solver is essentially a concrete and optimized implementation in CVC4 of the proof procedure, as we explain in Sect. 6. In Sect. 7, we present an experimental evaluation of our implementation against other major string solvers. We conclude in Sect. 8, mentioning several areas of future work.

This paper is a considerably extended and revised version of one presented at CAV 2014 [18]. Major differences include full proofs of our theoretical results, and an updated experimental evaluation with new solvers and more recent versions of previously existing ones. As in that work, however, we focus mostly on solving equality and disequality constraints over string terms together with arithmetic constraints over their length. The part of the calculus used to reason about membership predicates is substantial and different enough to deserve a separate treatment [19].

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 [14] 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 [15]. 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 [13]. Generally speaking, there are two kinds of automata-based approaches: one where each transition in the automaton represents a single character (e.g., [10,34]), and one where each transition represents a set of characters (e.g., [15,32,33]). 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 [12,17].

A different class of solvers is based on reducing string constraints to constraints in other theories. A rather successful representative of this approach is the Hampi solver [16], which has been 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 [29], a white-box test generation tool, targets path feasibility problems for programs using the .NET string library [4]. 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 [27] 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. It has been used in the WebBlaze tool to find new vulnerabilities in Javascript programs in cases where a bit-vector encoding would not be suitable [27].

The Java string analyzer (JSA) [7] 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 by a regular one, which is then encoded as a multi-level automaton.

Compared to our work, JSA focuses on Java string analysis, approximation, and automaton conversion, while our approach is independent from the original programming language, and solves string constraints primitively without approximation. PASS [17] combines ideas from automata and SMT. As with 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.

Recent concurrent work by others has resulted in three systems that are more closely related to our approach and its implementation in CVC4. The first is Z3- STR [35], a string solver developed as an extension of the Z3 SMT solver [8] 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. In contrast with CVC4, its main idea is to have Z3 treat string function and predicate symbols as uninterpreted but monitor the inferences of Z3's equality solver, generating and passing 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 not refutation complete since 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 notable 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. The third difference is that when a string variable appears on both sides of an equality, Z3- STR under-approximates candidate models and confines its search space to a finite set. If no candidate satisfies the constraints, Z3- STR reports unknown. In contrast, our approach reduces these sorts of constraints to *symbolic regular expressions*. This preserves all possible models and allows us to prove that our proof procedure is refutation sound (Theorem 1).

Another recent SMT-based string solver is S3 [31]. It is built as an extension of Z3- STR with support for additional string manipulating functions and regular expressions. The reduction rules for handling membership constraints in S3 are similar to the simplified lazy unrolling rules mentioned in this paper. This approach is refutation incomplete even over the restricted fragment of just membership constraints. That is the case also for the lazy unrolling rules we discuss later in this paper. However, CVC4 actually uses a different set of rules that make it complete over that fragment. A description of those rules is given in a separate paper [19]. A final difference with CVC4 is that neither S3 nor Z3- STR generate unsatisfiable cores for unsatisfiable problems.

The work most closely related to ours is by the authors of NORN, a string solver based on a recent calculus [1] for the same fragment of the theory of strings we discuss in this paper. According to its authors, the latest version of NORN [1] follows our approach by using the DPLL(T) architecture (see Sect. 6) to implement an efficient proof procedure for the calculus. A major feature of NORN is that it is a decision procedure over a large class of *acyclic* constraints identified by the authors in [1]. While as an SMT solver it supports only the theory of strings, it also contains a fixed-point engine used to prove safety properties of recursive programs encoded as Horn-like clauses.

[1] Personal communication. NORN was not publicly available at the time of this writing.

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 following notions: signature, term, literal, formula, free variable, interpretation, and satisfiability of a formula in an interpretation (see, e.g., [3] for more details). Let Σ be a many-sorted signature. 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 or formulas as expected.

If φ is a Σ-formula and $\mathcal{I}$ a Σ-interpretation, we write $\mathcal{I} \models \varphi$ if $\mathcal{I}$ satisfies φ. If t is a term, we denote by $t^{\mathcal{I}}$ the value of t in $\mathcal{I}$. 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 $\mathbf{I}$). 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 Σ-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. We write $\models_T \varphi$ as an abbreviation of $\emptyset \models_T \varphi$. We write $\Gamma \models \varphi$ to denote that Γ entails φ in the class of all Σ-interpretations. The set Γ is *satisfiable in* T if $\Gamma \not\models_T \bot$ where $\bot$ is the universally false atom. 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. When convenient, we will tacitly treat a finite set of formulas as the conjunction of its elements.

2 A theory of strings and regular language membership

We consider a parametric theory T_{SRL} of Strings with Regular language membership and Length constraints. The constraints are from a signature Σ_{SRL} with three sorts, Str, Int, and Lan, and an infinite set of variables for each sort. This theory, which is parametrized by a *finite* set $\mathcal{A}$ of *characters*, is essentially the theory of a single many-sorted structure, as its models differ only on how they interpret the variables. All models of T_{SRL} interpret Int as the set of integer numbers, Str as the language $\mathcal{A}^*$ of all words over the alphabet $\mathcal{A}$, and Lan as the power set of $\mathcal{A}^*$. The signature includes the following predicate and function symbols, summarized in Fig. 1: 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 (and so identified with it); a variadic function symbol $\mathsf{con} : \mathsf{Str} \times \cdots \times \mathsf{Str} \to \mathsf{Str}$, interpreted

Sort symbols:

$\mathsf{Str} \qquad \mathsf{Int} \qquad \mathsf{Lan}$

Function symbols:

$n : \mathsf{Int}$ for all $n \in \mathbb{N}$	$+ : \mathsf{Int} \times \mathsf{Int} \to \mathsf{Int}$	$- : \mathsf{Int} \to \mathsf{Int}$
$l : \mathsf{Str}$ for all $l \in \mathcal{A}^*$	$\mathsf{con} : \mathsf{Str} \times \cdots \times \mathsf{Str} \to \mathsf{Str}$	$\mathsf{len} : \mathsf{Str} \to \mathsf{Int}$
$\mathsf{set} : \mathsf{Str} \to \mathsf{Lan}$	$\mathsf{star} : \mathsf{Lan} \to \mathsf{Lan}$	$\ldots$

Predicate symbols:

$> : \mathsf{Int} \times \mathsf{Int} \qquad \mathsf{in} : \mathsf{Str} \times \mathsf{Lan}$

Fig. 1 The signature of T_{SRL}

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 $l \in \mathcal{A}^*$ to the language $\{l\}$; 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; and a suitable set of additional function symbols corresponding to regular expression operators such as language concatenation, conjunction, disjunction, and so on.[2]

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 linear 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 regular language constraint is a literal of the form $[\neg]s \;\mathsf{in}\; r$ where s is a string term and r is a regular expression. A T_{SRL}*-constraint* is a string, arithmetic or regular language constraint. We will denote entailment in T_{SRL} ($\models_{T_{\mathsf{SRL}}}$) more simply as $\models_{\mathsf{SRL}}$.

2.1 The satisfiability problem in T_{SRL}

We are interested in checking the satisfiability in T_{SRL} of finite sets of T_{SRL}-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., [11]). Some other sublanguages do have a decidable satisfiability problem. For instance, the satisfiability of word equations was proven decidable by Makanin [20] and then given a PSPACE algorithm by Plandowski [25], although that algorithm is highly impractical.

In this paper, we focus on practical solvers for the full language of T_{SRL}-constraints that, while incomplete and non-terminating in general, can efficiently solve 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 regular language constraints. The combination between the two solvers is achieved, Nelson–Oppen style, by exchanging equalities over shared terms, which however are not variables, as in traditional combination procedures [22, 30], but terms of the form $(\mathsf{len}\, x)$ where x is a variable.[3]

Like the theory T_{SRL}, our constraint solving method for it is parametric in the finite alphabet $\mathcal{A}$; the method works exactly the same way regardless of the alphabet as long as it is finite. In fact, the method can be modified slightly, by simply dropping the derivation rule Card from Fig. 7, so that it works also with infinite alphabets over problems that do not have membership constraints.

[2] We do not specify those additional symbols here because solving membership constraints is not the focus of this paper.

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

3 A calculus for T_{SRL}

In this section, we describe the essence of our combined solver for T_{SRL} 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 the description below, we focus only on the derivation rules that deal with string and arithmetic constraints. We have, additionally, developed a novel set of rules for processing regular language constraints using algebraic techniques. These are complex and sophisticated enough to deserve a dedicated description in a separate paper [19], and so are not covered here.

The rules of the calculus describe abstractly how the two subsolvers, respectively for arithmetic and for string and regular membership constraints, behave and cooperate. Since the arithmetic solver is completely standard, we provide only rules that formalize its interaction with the string solver. We give only one rule involving regular language constraints, to process membership constraints generated from word equations. The rest of the rules then formalize the part of the solver that processes string constraints.

The string constraint subsolver is built on top of a more or less standard congruence closure procedure for EUF, the theory of equality with uninterpreted functions, that can deduce general consequences (in that theory) of the current set of constraints. When computing the congruence closure of that set, this procedure essentially treats the string operators as if they were uninterpreted. The only significant difference is that it recognizes all string constants as denoting pairwise distinct elements of the string domain. The rest of the string solver contains rules that deduce, or guess, consequences specific to the theory of strings. For instance, it can deduce the equation $y \approx z$ from $\mathsf{con}(x, y) \approx \mathsf{con}(x, z)$ or from $\mathsf{con}(x_1, y) \approx \mathsf{con}(x_2, z)$ and $\mathsf{len}\, x_1 \approx \mathsf{len}\, x_2$. Deductions and guesses are actually made on *flat* forms of the terms involved instead of the actual terms. A flat form is an equivalent representation of a term as a flat concatenation of string constants and variables. The computation of flat forms proceeds bottom-up from sub-terms to super-terms and is such that all terms in the same equivalence class eventually get the same flat form, which at that point becomes a *normal* form for the whole class. For simplicity, and in fact also for efficiency, flat and normal forms are recomputed from scratch every time a new equation is added to the current set of constraints. When all normal forms have been computed, equivalence classes are partitioned into *buckets*, where each bucket is associated with a different string length. This partitioning aids in computing a concrete solution for the original problem, that is, a satisfying variable assignment mapping each variable to a string/integer constant.

3.1 Basic notions and assumptions

To describe our calculus formally, we adopt the following notational conventions. We will denote characters (i.e., elements of the alphabet $\mathcal{A}$) by the letter c, and string constants by l or the juxtaposition $c_1 \ldots c_n$ of their individual characters, with $c_1 \ldots c_n$ 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.

The calculus will maintain terms fully reduced with respect to the rewrite rules in Fig. 2, which can be shown to be terminating and confluent modulo the axioms of arithmetic. We denote by $t\!\downarrow$ the fully reduced form of a term t with respect to those rewrite rules. 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

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

Fig. 2 Rewrite rules for terms

form $\mathsf{con}(a_1, \ldots, a_n)$ with $n > 1$ and $a_1, \ldots, a_n$ atomic; if t is of integer sort, then $t{\downarrow}$ is an arithmetic term.

We will consider tuples $(s_1, \ldots, s_n)$ of string terms, with $n \geq 0$, and denote them by letters in bold font, with comma denoting tuple concatenation. For example, if $\mathbf{s} = (s_1, s_2)$ and $\mathbf{t} = (t_1, t_2, t_3)$ we will write $(\mathbf{s}, \mathbf{t})$ to denote the tuple $(s_1, s_2, t_1, t_2, t_3)$. Similarly, if u is a term, $(\mathbf{s}, u, \mathbf{t})$ denotes the tuple $(s_1, s_2, u, t_1, t_2, t_3)$. We will sometimes use *fully reduced* tuples $\mathbf{a}{\downarrow}$ of atomic terms obtained from an atomic term tuple $\mathbf{a}$ by dropping its empty string components and replacing adjacent string constants by the constant corresponding to their concatenation. For example, $(x, \epsilon, c_1, c_2c_3, y){\downarrow} = (x, c_1c_2c_3, y)$ and $(\epsilon){\downarrow} = ()$.

Definition 1 (*Congruence closure*) Let S be a finite 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

$$\widehat{S} = \{s \approx t \mid s, t \in \mathcal{T}_S,\ S \models s \approx t\} \cup$$
$$\{s \not\approx t \mid s, t \in \mathcal{T}_S,\ s' \not\approx t' \in S \cup L,\ S \models s \approx s' \wedge t \approx t' \text{ for some } s', t'\}$$

where $L = \{l_1 \not\approx l_2 \mid l_1, l_2 \text{ distinct string constants}\}$. □

Note that while $\widehat{S}$ is infinite, it only contains finitely many equalities. The congruence closure of S induces an equivalence relation $\mathbf{E}_S$ over $\mathcal{T}_S$ where two terms s, t are equivalent iff $s \approx t \in \widehat{S}$ (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.

The calculus applies to a finite set of T_{SRL}-constraints with the goal of determining its satisfiability in T_{SRL}. The rules of our calculus operate over data structures we call *configurations*.

Definition 2 (*Configurations*) A *configuration* is either the distinguished symbol unsat or a tuple of the form $\langle S, A, R, F, N, C, B\rangle$ where

– S, A, R are respectively a set of string, arithmetic, and regular language constraints;
– F is a set of pairs $s \mapsto \mathbf{a}$ where $s \in \mathcal{T}_{\mathsf{S}}$ and $\mathbf{a}$ is a tuple of atomic string terms;
– N is a set of pairs $e \mapsto \mathbf{a}$ where e is an equivalence class of $\mathbf{E}_S$ and $\mathbf{a}$ 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 $\mathbf{E}_S$. □

Informally, the sets S, A, R initially store the input problem and grow with additional constraints derived by the calculus; N eventually stores a *normal form* for each equivalence class in $\mathbf{E}_S$; 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 for which a flat form should not be computed, to prevent loops in the computation of their equivalence class' normal form; B eventually becomes a partition of the set of equivalence classes of $\mathbf{E}_S$ and is used to generate a satisfying assignment that maps equivalence classes in *different* buckets to string constants of *different* lengths, and different equivalence classes in the *same* bucket to different string constants of the *same* length.

Assumption 1 By standard transformations, one can convert any finite set of Σ_{SRL}-literals to an equisatisfiable set $S_0 \cup A_0 \cup R_0$, where S_0 is a set of string constraints, A_0 is a set of arithmetic constraints, and R_0 is a set of regular language constraints. As a consequence, without loss of generality, we define our calculus to start only with configurations of the form $\langle S_0, A_0, R_0, \emptyset, \emptyset, \emptyset, \emptyset \rangle$. For convenience, we also assume that

1. S_0 contains no trivial equations $t \approx t$;
2. S_0 contains an equation $x \approx t$ for each non-variable term $t \in \mathcal{T}_{S_0}$, where x is a variable of the same sort as t;[4]
3. all string variables occurring in A_0 also occur in S_0;[5]
4. all terms in $S_0 \cup A_0 \cup R_0$ are fully reduced with respect to the rewrite system of Fig. 2.

□

3.2 Derivation rules

The rules of the calculus are provided in Figs. 3, 4, 5, 6 and 7 in *guarded assignment form*, where fields S, A, R, F, N, C, and B store, in order, the components of a configuration $\langle S, A, R, N, C, B \rangle$. A derivation rule applies to a configuration K if all of the rule's premises hold for K *and* the resulting configuration is different from K. A rule's conclusion describes how each component of K is changed, if at all. In the rules, we write S, t as an abbreviation for $S \cup \{t\}$. Rules with two or more conclusions separated by the symbol $\parallel$ are non-deterministic branching rules.

For notational convenience, in rule premises and conclusions we treat any string constant l in a tuple of terms indifferently as an atomic 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 $c_1c_2c_3$ may be treated also as the tuples (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 $\approx$. When reading the rules, it helps to keep in mind that every non-variable string term in a configuration is equated to a variable in the S component.

We assume the availability of a terminating procedure for checking entailment in the theory of linear integer arithmetic ($\models_{\mathsf{LIA}}$) and one for computing congruence closures in the sense of Definition 1 and checking entailment in EUF ($\models$). Implementations of such procedures are available in most SMT solvers.

Proof procedures for the calculus maintain at all times a current configuration which they modify using one of the calculus' rules. As we will see, the calculus is such that the current configuration $\langle \mathsf{S}, \mathsf{A}, \mathsf{R}, \mathsf{F}, \mathsf{N}, \mathsf{C}, \mathsf{B} \rangle$ satisfies the following.

Invariant 1
1. All terms are fully reduced with respect to the rewrite system in Fig. 2.
2. F is a partial map from $\mathcal{T}_{\mathsf{S}}$ to fully reduced tuples of atomic terms.
3. N is a partial map from $\mathbf{E}_{\mathsf{S}}$ to fully reduced 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 that $\mathsf{S} \models_{\mathsf{SRL}} s \approx \mathsf{con}(a_1, \ldots, a_n)$.
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 (fully reduced) terms of the form $\mathsf{con}(\mathbf{a})$. □

[4] Such equations can always be added as needed using fresh variables without changing the satisfiability of the original problem.

[5] If x occurs only in A_0, necessarily in a term of the form $\mathsf{len}\, x$, the whole term can be replaced by a fresh arithmetic variable.

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

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

$$\text{A-Conflict}\ \frac{\mathsf{A} \models_{\mathsf{LIA}} \perp}{\mathsf{unsat}} \qquad \text{R-Star}\ \frac{s \text{ in } \mathsf{star}(\mathsf{set}\ t) \in \mathsf{R} \quad s \not\approx \epsilon \in \widehat{\mathsf{S}}}{\mathsf{S} := \mathsf{S},\ s \approx \mathsf{con}(t, z_{s,t}){\downarrow} \quad \mathsf{R} := \mathsf{R},\ z_{s,t} \text{ in } \mathsf{star}(\mathsf{set}\ t)}$$

Fig. 3 Rules for theory combination, arithmetic and regular language constraints. $z_{s,t}$ denotes a distinguished fresh variable for the pair s, t

$$\text{S-Cycle}\ \frac{t = \mathsf{con}\,(\boldsymbol{u}_1, s, \boldsymbol{u}_2) \quad t \in \mathcal{T}_{\mathsf{S}} \setminus \mathsf{C} \quad u \approx \epsilon \in \widehat{\mathsf{S}} \text{ for all } u \text{ in } (\boldsymbol{u}_1, \boldsymbol{u}_2)}{\mathsf{S} := \mathsf{S}, s \approx t \quad \mathsf{C} := (\mathsf{C}, t) \setminus \{s\}}$$

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

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

Fig. 4 Basic string derivation rules

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

3.2.1 Rule groups

We divide the rules into several groups here to facilitate their description. The first four rules in Fig. 3 describe the interaction between arithmetic reasoning and string reasoning, achieved via the propagation of entailed constraints in the shared language. Rule A-Conflict derives unsat if the arithmetic part of the problem is unsatisfiable. R-Star is the only rule for handling regular language constraints that we provide here, because the constraints it applies to can be generated, by rule F-Loop in Fig. 6, even if the initial configuration contains no regular language constraints.

The basic rules for string constraints are shown in Fig. 4. S-Conflict derives a contradiction when $\widehat{\mathsf{S}}$ contains a trivially unsatisfiable equality. S-Cycle equates a concatenation t of terms with one of them (s) when the remaining ones are all equivalent to ϵ. It also marks t, by adding it to the component C, as a term for which no flat form should be computed, which prevents the appearance of cycles in the flat from computation. In the absence of additional information, S-Split guesses whether two strings are equal or not, while L-Split guesses whether two string variables have the same length. Deciding on the equality of string variables and their length is important for constructing satisfying assignments.

The bulk of the work is done by the rules in Figs. 5 and 6. Those in Fig. 5 are used to compute an equivalent flat form (consisting of a sequence of atomic terms) for each non-variable term 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 the variables in e is chosen, arbitrarily,

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

$$\text{N-Form1}\ \frac{[x] \notin \mathcal{D}(\mathsf{N}) \quad [x] \not\subseteq \mathsf{C} \cup \mathcal{V}(\mathsf{S}) \\ \text{for all } t \in [x] \setminus (\mathsf{C} \cup \mathcal{V}(\mathsf{S}))\ t \in \mathcal{D}(\mathsf{F}) \text{ and } \mathsf{F}\,t = \boldsymbol{a}}{\mathsf{N} := \mathsf{N}, [x] \mapsto \boldsymbol{a}}$$

$$\text{N-Form2}\ \frac{[x] \notin \mathcal{D}(\mathsf{N}) \quad [x] \subseteq \mathsf{C} \cup \mathcal{V}(\mathsf{S})}{\mathsf{N} := \mathsf{N}, [x] \mapsto (x)} \qquad \text{Reset}\ \frac{}{\mathsf{F} := \emptyset \quad \mathsf{N} := \emptyset \quad \mathsf{B} := \emptyset}$$

Fig. 5 Normalization derivation rules. The letter l denotes a string constant

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

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

$$\text{F-Loop}\ \frac{\mathsf{F}\,s = (\boldsymbol{w}, x, \boldsymbol{u}_1) \quad \mathsf{F}\,t = (\boldsymbol{w}, v, \boldsymbol{v}_1, x, \boldsymbol{v}_2) \quad s \approx t \in \widehat{\mathsf{S}} \quad x \notin \mathcal{V}(v, \boldsymbol{v}_1) \quad \mathsf{S} \models \mathsf{con}(v, \boldsymbol{v}_1) \not\approx \epsilon}{\mathsf{S} := \mathsf{S},\ x \approx \mathsf{con}(z_2, z),\ \mathsf{con}(v, \boldsymbol{v}_1)\!\downarrow \approx \mathsf{con}(z_2, z_1),\ \mathsf{con}(\boldsymbol{u}_1)\!\downarrow \approx \mathsf{con}(z_1, z_2, \boldsymbol{v}_2) \\ \mathsf{R} := \mathsf{R}, z \text{ in } \mathsf{star}(\mathsf{set}\ \mathsf{con}(z_1, z_2)) \quad \mathsf{C} := (\mathsf{C}, t) \setminus \{s\}}$$

Fig. 6 Equality reduction rules. The letters z, z_1, z_2 denote distinct fresh variables

by N-Form2 as the normal form of e. Reset is meant to be applied after the set S changes, since in that case normal and flat forms and buckets may need to be recomputed.

Example 1 To see how normal forms are computed by the calculus, say that

$$\mathsf{S} = \{x \approx \mathsf{con}(y, z),\ z \approx \mathsf{con}(w, y)\}$$

initially, where x, y, z and w are all variables. The equivalence classes in $\widehat{\mathsf{S}}$ are $\{x, \mathsf{con}(y, z)\}$, $\{y\}$, $\{z, \mathsf{con}(w, y)\}$, and $\{w\}$. Assuming the other components of our configuration are empty, we may compute a total map N over these equivalence classes using the rules in Fig. 5. First, we may apply N-Form2 to add $[y] \mapsto (y)$ and $[w] \mapsto (w)$ to N. Then, we may concatenate these tuples with F-Form1 to add $\mathsf{con}(w, y) \mapsto (w, y)$ to F. After doing so, since a flat form exists for each non-variable term in $[z]$, using N-Form1 we may add $[z] \mapsto (w, y)$ to N. Likewise, we may apply F-Form1 to add $\mathsf{con}(y, z) \mapsto (y, w, y)$ to F, and N-Form1 to add $[x] \mapsto (y, w, y)$ to N. □

The first two rules of Fig. 6 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 Sect. 5 and Example 7 in that section for more details on F-Loop.

Example 2 For a more elaborate example of normal form computation that involves also the rules of Fig. 6, consider an initial configuration with

$$\mathsf{S} = \{x \approx \mathsf{con}(y, z),\ z \approx \mathsf{con}(w, y),\ x \approx \mathsf{con}(y, u, y)\} \text{ and } \mathsf{A} = \{\mathsf{len}\,u \approx \mathsf{len}\,w\}\,.$$

In this case, the equivalence class of x is $\{x, \mathsf{con}(y, z), \mathsf{con}(y, u, y)\}$. We could apply the steps in the previous example to add $\mathsf{con}(y, z) \mapsto (y, w, y)$ to F, and similarly to add

$$\textsf{D-Base}\ \frac{\begin{array}{c} s \in \mathcal{T}_{\mathsf{S}} \quad s : \mathsf{Str} \\ \mathsf{S} \models \mathsf{len}\, s \not\approx \mathit{len}_b \text{ for all } b \in \mathsf{B} \end{array}}{\mathsf{B} := \mathsf{B}, \{[s]\}} \qquad \textsf{Card}\ \frac{b \in \mathsf{B} \quad |b| > 1}{\mathsf{A} := \mathsf{A}, \mathit{len}_b > \lfloor \log_{|\mathcal{A}|}(|b| - 1) \rfloor}$$

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

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

Fig. 7 Disequality reduction rules. Letters z_1, z_2 denote distinct fresh variables. For each bucket $b \in \mathsf{B}$, len_b denotes a unique term $(\mathsf{len}\, z)$ where $[z] \in b$. The operator $|\_|$ is the set cardinality operator

$\mathsf{con}(y, u, y) \mapsto (y, u, y)$ to F. However, notice that N-Form cannot be applied to the equivalence class $[x]$, since it contains two terms having different flat forms. When this occurs, we may use the rules in Fig. 6 to infer additional equalities to resolve this difference. In this example, we may apply F-Unify to add $u \approx w$ to S, which will cause the equivalence classes of u and w to merge. As a consequence, we apply Reset to recompute flat and normal forms for the terms in S. After that, we obtain identical flat forms for $\mathsf{con}(y, z)$ and $\mathsf{con}(y, u, y)$, which allows us to apply N-Norm1 and obtain a normal form for $[x]$. □

The rules in Fig. 7 are used to partition the equivalence classes of terms of sort Str into buckets. The partitioning is based on the expected length of the value that each equivalence class will eventually be given by a satisfying assignment. Once the relationship between these lengths is known, different equivalence classes go into different buckets (using D-Base) if their respective terms are constrained to have different lengths. Otherwise, if their terms are constrained to have the same length, they go into the same bucket (using D-Add), but only if we can tell they cannot have the same value. D-Split is used to align the normal forms so that each equivalence class can be either added to an existing bucket with D-Base or given its bucket with D-Add. 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 of that length. Card adds a constraint guaranteeing that the chosen n_b is big enough to allow for the existence of enough distinct strings of length n_b.

Example 3 As an example of the application of the rules in Fig. 7, consider a configuration where $\widehat{\mathsf{S}}$ contains three equivalence classes $[x]$, $[y]$, and $[z]$, and suppose we have computed the following normal forms for each of these classes:

$$\mathsf{N}\,[x] = (w_1, w_2, w_3) \qquad \mathsf{N}\,[y] = (w_1, w_4, w_5) \qquad \mathsf{N}\,[z] = (w_1, w_6)\ .$$

Furthermore, assume that $\mathsf{S} \models \mathsf{len}\, x \approx \mathsf{len}\, y$ and $\mathsf{S} \models \mathsf{len}\, x \not\approx \mathsf{len}\, z$. We may use the rules in Fig. 7 to partition the equivalence classes of $\widehat{\mathsf{S}}$. Assuming B is initially empty, we may apply D-Base to add $\{[x]\}$ and $\{[z]\}$ to B. Since $\mathsf{S} \models \mathsf{len}\, x \approx \mathsf{len}\, y$, in order to add $[y]$ to B we must ensure that the precondition of D-Add is met. Assume that $\mathsf{S} \models \mathsf{len}\, w_2 \approx \mathsf{len}\, w_4$ and $w_2 \not\approx w_4 \in \widehat{\mathsf{S}}$. We may apply D-Add to obtain the bucket $b = \{[x], [y]\}$. Assuming the cardinality of our alphabet is 256, we may apply Card to b to add the constraint $\mathsf{len}\, x > 0$, stating that since there are two distinct equivalence classes of strings ($[x]$ and $[y]$) having the same length, that length must be at least 1. □

3.3 Derivation trees and derivations

Because of the presence of non-deterministic rules, derivations in the calculus construct derivation trees.

Definition 3 (*Derivation tree*) A *derivation tree* is a tree where: (i) each node is a configuration whose children (if any) are obtained by applying to it one of the derivation rules, and (ii) the root node, which we call the *initial* configuration, satisfies Assumption 1. $\square$

A derivation tree is *closed* if all of its branches (are finite and) end with the unsat configuration. A derivation tree *derives* from a derivation tree T if it is obtained from T by the application of *exactly one* of the derivation rules to one of T's leaves. A configuration is *derivable* if it occurs in a derivation tree; it *derives* from a configuration K if it is a child of K in a derivation tree.

Definition 4 (*Derivation*) A *derivation* is a (possibly infinite) sequence $(T_i)_{i \in I}$ of derivation trees with $I \subseteq \mathbb{N}$, such that T_0 is a one-node tree whose root is an initial configuration and T_i derives from T_{i-1} for all positive $i \in I$. $\square$

When talking about the process of constructing a derivation, we will refer to the most recently derived configuration as the *current* configuration.

4 Partial correctness of the calculus

In this section, we formalize the main correctness properties of our calculus, namely that it is refutation and solution sound. Since a string solver for the theory T_{SRL} can be built as a specific proof procedure for this calculus, as we illustrate in Sect. 6, such a solver immediately inherits those properties. This means in particular that when the 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 regular language constraints*, as those constraints are not the focus of this work.

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

Proof Immediate consequence of the fact that in each of the rewrite rules of Fig. 2, the left-hand side is equivalent in T_{SRL} to the right hand side. $\square$

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

Proof Let $K = \langle S, A, R, F, N, C, B \rangle$ be a derivable configuration. If K is an initial configuration, Invariant 1 holds trivially by our assumptions on such configurations. We show that Properties 1 through 6 of Invariant 1 are preserved by each application of a derivation rule in our calculus. For each property, we consider only those rules for which the preservation of the property is not immediate.

(Property 1) To see that this property is preserved, observe that every new term introduced by the conclusion of a rule (as in F-Split, F-Loop, D-Split, and so on) is either already fully reduced (because it is, for instance, an atomic term or the concatenation of a variable with other atomic terms) or gets reduced explicitly with the rewrite rules.

(Properties 2 and 1) The rules in Fig. 5 are the only ones that modify F. We note that their premises ensure that entries can be added to F only for terms from $\mathcal{T}_{\mathsf{S}}$ not in the current domain of F. Also note that in $\mathsf{F\text{-}Form1}$, we know that the new tuple is made up of atomic terms because of the premises and Property 3. Finally, new tuples are explicitly fully reduced in each rule modifying F. The argument for Property 3 is similar. We note only that in $\mathsf{N\text{-}Form2}$, a tuple consisting of a single variable is always fully reduced.

(Property 4) The rules in Fig. 5 are the only ones that may impact this property. For $\mathsf{F\text{-}Form1}$, assuming that Property 4 is satisfied by the premise configuration, we have that $\mathsf{S} \models_{\mathsf{SRL}} t_1 \approx \mathsf{con}(\mathbf{s}_1) \wedge \ldots \wedge t_n \approx \mathsf{con}(\mathbf{s}_n)$. It is easy to see that then S T_{SRL}-entails the equality chain $\mathsf{con}(t_1, \ldots, t_n) \approx \mathsf{con}(\mathsf{con}(\mathbf{s}_1), \ldots, \mathsf{con}(\mathbf{s}_n)) \approx \mathsf{con}(\mathbf{s}_1, \ldots, \mathbf{s}_n) \approx \mathsf{con}((\mathbf{s}_1, \ldots, \mathbf{s}_n){\downarrow})$. For $\mathsf{F\text{-}Form2}$, it is enough to observe that $(\epsilon){\downarrow} = ()$ and $(l){\downarrow} = (l)$ for all other string constants l and that $\models_{\mathsf{SRL}} \epsilon \approx \mathsf{con}()$ and $\models_{\mathsf{SRL}} l \approx \mathsf{con}(l)$. For $\mathsf{N\text{-}Form1}$, by assumption of the invariant on the premise configuration we have that $\mathsf{S} \models_{\mathsf{SRL}} t \approx \mathsf{con}(\mathsf{F}\, t)$ for some $t \in [x]$ with $\mathbf{a} = \mathsf{F}\, t$ satisfying Property 4. Hence, $\mathsf{S} \models_{\mathsf{SRL}} x \approx \mathsf{con}(\mathbf{a})$. Since $\mathsf{N}\,[x] = \mathbf{a}$ after applying the rule, we have that $\mathsf{S} \models_{\mathsf{SRL}} x \approx \mathsf{con}(\mathsf{N}\,[x])$ and so $\mathsf{S} \models_{\mathsf{SRL}} s \approx \mathsf{con}(\mathsf{N}\,[x])$ for all $s \in [x]$. Finally, for $\mathsf{N\text{-}Form2}$, note that $\mathsf{S} \models_{\mathsf{SRL}} x \approx \mathsf{con}(x)$. It follows that $\mathsf{S} \models_{\mathsf{SRL}} s \approx \mathsf{con}(x)$ for all $s \in [x]$.

(Property 5) For $\mathsf{D\text{-}Base}$, note that the rule creates a new bucket containing equivalence class $[s]$ only when $\mathsf{S} \models \mathsf{len}\, s \not\approx len_b$ for all $b \in \mathsf{B}$. For $\mathsf{D\text{-}Add}$, note that the rule adds an equivalence class $[s]$ to an existing bucket b only when $\mathsf{S} \models_{\mathsf{SRL}} \mathsf{len}\, s \approx len_b$. Assuming Property 5 holds on the premise configuration, we have then that $\mathsf{S} \models \mathsf{len}\, s \approx \mathsf{len}\, t$ for all $[t] \in b$ and $\mathsf{S} \not\models \mathsf{len}\, s \approx \mathsf{len}\, t$ for all $[t]$ in other buckets of B.

(Property 6) For $\mathsf{S\text{-}Cycle}$, observe that t has the form $\mathsf{con}(\mathbf{a})$ by the premise of the rule. For $\mathsf{F\text{-}Loop}$, observe that since $\mathsf{F}\, t$ contains a variable, x, it must have been constructed by an application of $\mathsf{F\text{-}Form1}$, which means that t has the form $\mathsf{con}(\mathbf{a})$. In both cases, because t appears in S, it is fully reduced by Property 1. □

4.1 Refutation soundness

The first main property is that our calculus is *refutation sound*, meaning that when it derives a refutation of $\langle S_0, A_0, \emptyset, \emptyset, \emptyset, \emptyset, \emptyset \rangle$, then $S_0 \cup A_0$ is unsatisfiable in T_{SRL}. Roughly speaking, we prove this by showing that applications of the rules of the calculus preserve the models of the configuration they modify. This is straightforward for most rules of the calculus, with the exception of $\mathsf{F\text{-}Loop}$ for which we need two auxiliary results first.

In the following lemma we consider elements of the word algebra $\mathcal{A}^*$; we use juxtaposition to denote word concatenation and write l^n for the n-fold concatenation of word l with itself.

Lemma 3 *Let $l, l_1, l_2 \in \mathcal{A}^*$ with $l_1 \neq \epsilon$ and $l_2 \neq \epsilon$, such that $l\, l_2 = l_1 l$. Then $l = k_2(k_1k_2)^n$, $l_2 = k_1k_2$, and $l_1 = k_2k_1$ for some $k_1, k_2 \in \mathcal{A}^*$ and $n \geq 0$.*

Proof Overloading the notation, let $|\_|$ also denote word length. We prove the claim by induction on $|l|$.

($0 \leq |l| \leq |l_1|$) In this case, $l_1 = l\, k_1$ for some $k_1 \in \mathcal{A}^*$. The claim then follows by choosing $k_2 = l$ and $n = 0$.

($|l_1| < |l|$) In this case, $l = l_1 l'$ for some $l' \in \mathcal{A}^*$ with $|l'| < |l|$. Moreover, $l_1 l' l_2 = l_1 l_1 l'$ and so $l' l_2 = l_1 l'$. By the induction hypothesis, $l' = k_2(k_1k_2)^{n'}$, $l_2 = k_1k_2$, and $l_1 = k_2k_1$ for some $k_1, k_2 \in \mathcal{A}^*$ and $n' \geq 0$. Therefore, $l = l_1 l' = k_2k_1k_2(k_1k_2)^{n'} = k_2(k_1k_2)^{n'+1}$. The claim immediately follows by choosing $n = n' + 1$. □

Lemma 4 *If $K' = \langle S', A, R', F, N, C', B \rangle$ is the result of applying* $\mathsf{F\text{-}Loop}$ *to a configuration $K = \langle S, A, R, F, N, C, B \rangle$, then $S' \cup R'$ and $S \cup R$ are equisatisfiable in T_{SRL}.*

Proof It is immediate that every interpretation satisfying $S' \cup R'$ satisfies $S \cup R$ as well. For the other direction, assume that in configuration K the premise of $\mathsf{F\text{-}Loop}$ holds for s and t, where $F\,s = (\mathbf{w}, x, \mathbf{u}_1)$ and $F\,t = (\mathbf{w}, v, \mathbf{v}_1, x, \mathbf{v}_2)$. Since $s \approx t \in \widehat{S}$ and due to Property 4 of Invariant 1, we have that $S \models_{\mathsf{SRL}} \mathsf{con}(x, \mathbf{u}_1) \approx \mathsf{con}(v, \mathbf{v}_1, x, \mathbf{v}_2)$. Consider any model $\mathcal{I}$ of T_{SRL} that satisfies $S \cup R$. By the above, we have that

$$x^{\mathcal{I}}\mathbf{u}_1^{\mathcal{I}} = l_1 x^{\mathcal{I}}\mathbf{v}_2^{\mathcal{I}} \tag{1}$$

where $l_1 = v^{\mathcal{I}}\mathbf{v}_1^{\mathcal{I}}$.

Note that, trivially, $x^{\mathcal{I}}$ cannot have more characters than $l_1 x^{\mathcal{I}}$ and that $l_1 \neq \epsilon$ by the premises of $\mathsf{F\text{-}Loop}$. From (1), we then have that there is some word $l_2 \neq \epsilon$ such that $x^{\mathcal{I}} l_2 = l_1 x^{\mathcal{I}}$ and $\mathbf{u}_1^{\mathcal{I}} = l_2 \mathbf{v}_2^{\mathcal{I}}$. By Lemma 3, $x^{\mathcal{I}} = k_2 k$, $l_2 = k_1 k_2$ and $l_1 = k_2 k_1$ for some $k_1, k_2 \in \mathcal{A}^*$, $k = (k_1 k_2)^n$, and $n \geq 0$.

Let z, z_1, z_2 be the fresh variables introduced by $\mathsf{F\text{-}Loop}$ into K', as shown in the conclusion of the rule. Since these variables do not occur in K, we can assume with no loss of generality by definition of T_{SRL} that $z^{\mathcal{I}} = k$, $z_1^{\mathcal{I}} = k_1$ and $z_2^{\mathcal{I}} = k_2$. To prove the claim, it is enough to show that $\mathcal{I}$ satisfies the constraints (i) $x \approx \mathsf{con}(z_2, z)$, (ii) $\mathsf{con}(v, \mathbf{v}_1) \approx \mathsf{con}(z_2, z_1)$, (iii) $\mathsf{con}(\mathbf{u}_1) \approx \mathsf{con}(z_1, z_2, \mathbf{v}_2)$, and (iv) $z \in \mathsf{star}(\mathsf{set}\,\mathsf{con}(z_1, z_2))$ introduced by the rule:

(i) $x^{\mathcal{I}} = k_2 k = z_2^{\mathcal{I}} z^{\mathcal{I}} = \mathsf{con}(z_2, z)^{\mathcal{I}}$.
(ii) $\mathsf{con}(v, \mathbf{v}_1)^{\mathcal{I}} = v^{\mathcal{I}}\mathbf{v}_1^{\mathcal{I}} = l_1 = k_2 k_1 = z_2^{\mathcal{I}} z_1^{\mathcal{I}} = \mathsf{con}(z_2, z_1)^{\mathcal{I}}$.
(iii) $\mathsf{con}(\mathbf{u}_1)^{\mathcal{I}} = \mathbf{u}_1^{\mathcal{I}} = l_2 \mathbf{v}_2^{\mathcal{I}} = k_1 k_2 \mathbf{v}_2^{\mathcal{I}} = z_1^{\mathcal{I}} z_2^{\mathcal{I}} \mathbf{v}_2^{\mathcal{I}} = \mathsf{con}(z_1, z_2, \mathbf{v}_2)^{\mathcal{I}}$.
(iv) $z^{\mathcal{I}} = k = (k_1 k_2)^n = (z_1^{\mathcal{I}} z_2^{\mathcal{I}})^n \in \mathsf{star}(\mathsf{set}\,\mathsf{con}(z_1, z_2))^{\mathcal{I}}$. □

Definition 5 (*Satisfiable configuration*) A configuration $\langle S, A, R, F, N, C, N\rangle$ is *satisfiable* if $S \cup A \cup R$ is satisfiable in a model of T_{SRL}. A configuration is *unsatisfiable* if it is the unsat configuration or is not satisfiable.

Lemma 5 *If a non-leaf node in a derivation tree is satisfiable then one of its children is satisfiable.*

Proof Let $K = \langle S, A, R, F, N, C, N\rangle$ be a non-leaf node in a derivation tree, and let $\mathcal{I}$ be a model of T_{SRL} that satisfies K. We prove the claim by cases, depending on the rule applied to K to derive its child(ren).

($\mathsf{S\text{-}Conflict}$, $\mathsf{A\text{-}Conflict}$) Since the conclusion of these rules is unsatisfiable, it is enough to argue they only apply to unsatisfiable configurations. This is, however, immediate.

($\mathsf{Len\text{-}Split}$, $\mathsf{S\text{-}Split}$, $\mathsf{L\text{-}Split}$) For each of these cases, $S \cup A \cup R$ changes by the addition of some constraint in one branch and its negation in the other branch. It follows that $\mathcal{I}$ satisfies one of K's two children.

($\mathsf{R\text{-}Star}$) Let s, t and $z_{s,t}$ be the terms mentioned in this rule. By assumption on $\mathcal{I}$ and the premise, we have that $s^{\mathcal{I}} = (t^{\mathcal{I}})^{n+1} = t^{\mathcal{I}}(t^{\mathcal{I}})^n$ for some $n \geq 0$. Since $z_{s,t}$ does not occur in K, we can assume with no loss of generality that $z_{s,t}^{\mathcal{I}} = (t^{\mathcal{I}})^n$. Hence, $s^{\mathcal{I}} = t^{\mathcal{I}} z_{s,t}^{\mathcal{I}}$ and $z_{s,t}^{\mathcal{I}} \in (t^{\mathcal{I}})^*$. It follows that $\mathcal{I}$ satisfies the conclusion of the rule.

($\mathsf{S\text{-}Cycle}$) Let $t = \mathsf{con}\,(\mathbf{v}_1, s, \mathbf{v}_2)$ as in the premise of this rule. By assumption of $\mathcal{I}$ and the premise, we have that $t^{\mathcal{I}} = (\mathsf{con}\,\mathbf{v}_1)^{\mathcal{I}} s^{\mathcal{I}} (\mathsf{con}\,\mathbf{v}_2)^{\mathcal{I}} = \epsilon s^{\mathcal{I}} \epsilon = s^{\mathcal{I}}$. Thus, $\mathcal{I} \models s \approx t$.

($\mathsf{F\text{-}Loop}$) By Lemma 4.

($\mathsf{F\text{-}Unify}$, $\mathsf{F\text{-}Split}$) Let $s, t, \mathbf{w}, u, \mathbf{u}_1, v, \mathbf{v}_1$ be the terms mentioned in the premise of these rules. By Property 4 of Invariant 1 for s and t, we have that $s^{\mathcal{I}} = (F\,s)^{\mathcal{I}} = \mathbf{w}^{\mathcal{I}} u^{\mathcal{I}} \mathbf{u}_1^{\mathcal{I}}$ and $t^{\mathcal{I}} = (F\,t)^{\mathcal{I}} = \mathbf{w}^{\mathcal{I}} v^{\mathcal{I}} \mathbf{v}_1^{\mathcal{I}}$. With $\mathsf{F\text{-}Unify}$, since $S \models_{\mathsf{SRL}} s \approx t \wedge \mathsf{len}\,u \approx \mathsf{len}\,v$, we have that $u^{\mathcal{I}} = v^{\mathcal{I}}$ and so $\mathcal{I} \models u \approx v$. With $\mathsf{F\text{-}Split}$, since $S \models_{\mathsf{SRL}} s \approx t \wedge \mathsf{len}\,u \not\approx \mathsf{len}\,v$, it must be that $v^{\mathcal{I}}$ is a prefix of $u^{\mathcal{I}}$ or vice versa. In the first case, $u^{\mathcal{I}} = v^{\mathcal{I}}k$ for some word k. Since z does not occur in K, we can assume that $z^{\mathcal{I}} = k$. Hence, $\mathcal{I} \models u \approx \mathsf{con}(v, z)$. The other case is similar.

(Card) Let b be as in the premise of the rule, and suppose $b = \{[s_1], \ldots, [s_n]\}$ for some $n > 1$. Let m be the cardinality of the alphabet $\mathcal{A}$. By a simple inductive argument based on $\mathsf{D\text{-}Base}$ and $\mathsf{D\text{-}Add}$, one can show that $s_1^{\mathcal{I}}, \ldots, s_n^{\mathcal{I}}$ all have length $len_b^{\mathcal{I}}$ and are all distinct. This means that $m^{len_b^{\mathcal{I}}} > n - 1$. Hence, $\mathcal{I} \models len_b > \lfloor \log_m (n-1) \rfloor$.

($\mathsf{D\text{-}Split}$) The argument is similar to that for $\mathsf{F\text{-}Split}$ but based on the fact that, because they have different lengths, either u is longer than v (first branch) or v is longer than u (second branch).

(Remaining rules) Immediate. □

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_{SRL}.*

Proof Assume that $\langle S_0, A_0, \emptyset, \emptyset, \emptyset, \emptyset, \emptyset \rangle$ has a closed derivation tree T. It is enough to prove by structural induction on T that each of its subtrees has an unsatisfiable configuration at its root.

(Base case) Immediate since all of T's leafs are the configuration unsat.

(Inductive step) Let K be the root of a non-leaf subtree of T. All children of K are the root of a subtree of T. By inductive hypothesis, they are unsatisfiable. It follows by the contrapositive of Lemma 5 that K is unsatisfiable. □

4.2 Solution soundness

The calculus is also *solution sound* in the following sense: any saturated configuration (see next) in a derivation tree determines a variable assignment that is a solution of the original problem—the one stored in the root of the tree.

Definition 6 (*Saturated configuration*) A configuration $\langle S, A, R, F, N, C, B \rangle$ in a derivation is *saturated* if (i) no rule of the calculus other than Reset applies to it, (ii) N is a total map over $\mathbf{E}_S$, and (iii) B is a partition of $\mathbf{E}_S$. □

A branch in a derivation tree is *saturated* if it contains a saturated configuration. When constructing a derivation, a proof procedure can stop as soon as it generates a saturated configuration since such a configuration is a witness to the satisfiability of the original problem. In particular, for a saturated configuration $K = \langle S, A, R, F, N, C, B \rangle$, it is possible to construct a model $\mathcal{I}_K$ of T_{SRL} that satisfies $S \cup A \cup R$. We describe this model construction below and then prove that the obtained model satisfies $S \cup A \cup R$. While our main result holds for arbitrary derivations of K, to simplify the proof we will assume that the rule $\mathsf{F\text{-}Loop}$ was never applied in the given derivation.[6] This means in particular that the component R of K is empty if the corresponding component in the initial configuration is also empty.

We fix a saturated configuration $K = \langle S, A, R, F, N, C, B \rangle$ with $R = \emptyset$ for the rest of the section, together with a derivation of K from an initial configuration of the form $\langle S_0, A_0, \emptyset, \emptyset, \emptyset, \emptyset, \emptyset \rangle$.

Lemma 6 *The following hold for configuration K.*

1. $A \models_{\mathsf{LIA}} (\mathsf{len}\, t){\downarrow} \approx (\mathsf{len}\,(\mathsf{con}\,(F\, t))){\downarrow}$ *for each $t \in \mathcal{D}(F)$;*

[6] $\mathsf{F\text{-}Loop}$ introduces regular expression which are not the focus of this work.

2. $A \models_{\mathsf{LIA}} \mathsf{len}\, x \approx (\mathsf{len}\,(\mathsf{con}\,(N\,[x])))\!\downarrow$ *for each* $[x] \in \mathcal{D}(N)$*;*
3. *For each* $x \in \mathcal{V}(S)$*, if* $N\,[x] = (a_1, \ldots, a_n)$ *with* $n > 1$ *then* $A \models_{\mathsf{LIA}} \mathsf{len}\, x > (\mathsf{len}\, a_i)\!\downarrow$ *for* $i = 1, \ldots, n$*.*

Proof We prove Point 1. and 2. by simultaneous induction on the derivation of K.

For Point 1., if t was added to $\mathcal{D}(F)$ by F-Form2, the claim is immediate. If t was added to $\mathcal{D}(F)$ by F-Form1, we have that $t = \mathsf{con}(t_1, \ldots, t_m)$ for some $t_1, \ldots, t_m$. For $i = 1, \ldots, m$, let x_i be a string variable such that $t_i \in [x_i]$ and let $\mathbf{s}_i = N\,[x_i]$. Note that each x_i exists by our assumption that every equivalence class contains a variable and each $\mathbf{s}_i$ exists because the configuration is saturated with respect to the rules of Fig. 5. Also note that, by the premises of F-Form1, each $\mathbf{s}_i$ was added to $\mathcal{D}(N)$ before t was added to $\mathcal{D}(F)$. By inductive hypothesis then, $A \models_{\mathsf{LIA}} (\mathsf{len}\, x_i)\!\downarrow \approx (\mathsf{len}\,(\mathsf{con}\, \mathbf{s}_i))\!\downarrow$ for $i = 1, \ldots, m$, and thus $A \models_{\mathsf{LIA}} (\mathsf{len}\, t_1)\!\downarrow + \cdots + (\mathsf{len}\, t_m)\!\downarrow \approx (\mathsf{len}\,(\mathsf{con}\, \mathbf{s}_1))\!\downarrow + \cdots + (\mathsf{len}\,(\mathsf{con}\, \mathbf{s}_m))\!\downarrow$. The claim then follows form the fact that $\models_{\mathsf{LIA}} (\mathsf{len}\, t)\!\downarrow \approx (\mathsf{len}\, t_1)\!\downarrow + \cdots + (\mathsf{len}\, t_m)\!\downarrow$ and $\models_{\mathsf{LIA}} (\mathsf{len}\, \mathsf{con}(N\,[t]))\!\downarrow \approx (\mathsf{len}\,(\mathsf{con}\, \mathbf{s}_1))\!\downarrow + \cdots + (\mathsf{len}\,(\mathsf{con}\, \mathbf{s}_m))\!\downarrow$.

For Point 2., if $[x]$ was added to $\mathcal{D}(N)$ by N-Form2, the claim is immediate. In the case that $[x]$ was added to $\mathcal{D}(N)$ by N-Form1, we have that $N\,[x] = F\,t$ for some term $t \in [x]$, that is, some term t such that $x \approx t \in \widehat{S}$. Since the configuration is saturated with respect to Len, we have $A \models_{\mathsf{LIA}} (\mathsf{len}\, x)\!\downarrow \approx (\mathsf{len}\, t)\!\downarrow$. By the premises of N-Form1, t was added to $\mathcal{D}(F)$ before $[x]$ was added to $\mathcal{D}(N)$. It follows by inductive hypothesis that $A \models_{\mathsf{LIA}} (\mathsf{len}\, t)\!\downarrow \approx (\mathsf{len}\,(\mathsf{con}\,(F\,t)))\!\downarrow$, and thus $A \models_{\mathsf{LIA}} (\mathsf{len}\, x)\!\downarrow \approx (\mathsf{len}\,(\mathsf{con}\,(N\,[x])))\!\downarrow$.

To show Point [3.], assume $N\,[x] = (a_1, \ldots, a_n)$ with $n > 1$. By Property 4 of Invariant 1, and by saturation of K with respect to Len, Len-Split and S-Conflict, $A \models_{\mathsf{LIA}} \mathsf{len}\, a_i > 0$ for all $i = 1, \ldots n$. By Point 2., $A \models_{\mathsf{LIA}} \mathsf{len}\, x \approx (\mathsf{len}\, a_1)\!\downarrow + \cdots + (\mathsf{len}\, a_n)\!\downarrow$. It follows that $A \models_{\mathsf{LIA}} \mathsf{len}\, x > (\mathsf{len}\, a_i)\!\downarrow$ for $i = 1, \ldots, n$. □

4.2.1 Model construction

We now show how to construct a model $\mathcal{I}$ of T_{SRL} from the saturated configuration K. Since all models of T_{SRL} interpret the function and predicate symbols of T_{SRL} in the same way, we only need to specify how $\mathcal{I}$ interprets variables. We focus on the variables in K as we can define $\mathcal{I}$ arbitrarily over those not in K.

(Integer variables) Observe that A is satisfiable in T_{SRL}. Otherwise as one can easily argue, it would be LIA-unsatisfiable, making A-Conflict applicable to K. Let $\mathcal{I}$ be a model of T_{SRL} that satisfies A. Note that $\mathcal{I}$ need not satisfy S; however, since K is saturated with respect to Len and Len-Split, it must interpret every term of the form $\mathsf{len}\, x$ in A as some non-negative integer n and x as a string constant of length n. We can choose $\mathcal{I}$ so that, for all distinct string variables x, y of K,

$$(\mathsf{len}\, x)^{\mathcal{I}} = (\mathsf{len}\, y)^{\mathcal{I}} \ \text{ iff } \ S \models \mathsf{len}\, x \approx \mathsf{len}\, y\ . \tag{2}$$

In fact, if $S \models \mathsf{len}\, x \approx \mathsf{len}\, y$ then $A \models_{\mathsf{LIA}} \mathsf{len}\, x \approx \mathsf{len}\, y$ by saturation of K with respect to A-Prop. Hence, $(\mathsf{len}\, x)^{\mathcal{I}}$ must equal $(\mathsf{len}\, y)^{\mathcal{I}}$. If, instead, $S \not\models \mathsf{len}\, x \approx \mathsf{len}\, y$ then it must be that $A \not\models_{\mathsf{LIA}} \mathsf{len}\, x \approx \mathsf{len}\, y$ as otherwise, by saturation with respect to S-Prop and L-Split, rule S-Conflict would apply. Hence, it is possible to have $(\mathsf{len}\, x)^{\mathcal{I}} \neq (\mathsf{len}\, y)^{\mathcal{I}}$.

We fix $\mathcal{I}$'s interpretation of the integer variables but redefine its interpretation of the string variables in K as follows.

(String variables) By point (iii) in the definition of saturated configuration, for every string variable of K its equivalence class is in some bucket of B, so let us consider those variables by bucket. By construction of $\mathcal{I}$ and Property 5 of Invariant 1, we have that $len_b^{\mathcal{I}} \neq len_{b'}^{\mathcal{I}}$

for all pairs of distinct buckets $b, b' \in B$.[7] Let $b_1, \ldots, b_k$ be an enumeration of B such that $len^{\mathcal{I}}_{b_1} < \cdots < len^{\mathcal{I}}_{b_k}$ and let $i \in \{1, \ldots, k\}$. We redefine the interpretation of the string variables of K by induction on i. Specifically for each equivalence class $e \in b_i$ we do the following depending of the value $N\, e$.[8]

1. $N\, e = (l)$ for some string constant l. We modify $\mathcal{I}$ so that $y^{\mathcal{I}} = l$ for all variables $y \in e$.
2. $N\, e = (a_1, \ldots, a_n)$ with $n > 1$. From Lemma 6([3.]), we can conclude that the variables that (necessarily) occur in $(a_1, \ldots, a_n)$ belong to equivalence classes in buckets preceding b_i in the enumeration above and so i must be greater than 1. By induction hypothesis, we have already redefined $\mathcal{I}$ for those variables. We then modify $\mathcal{I}$ further so that $y^{\mathcal{I}} = \mathsf{con}(a_1, \ldots, a_n)^{\mathcal{I}}$ for all $y \in e$.
3. $N\, e = (u)$ for some variable u. We modify $\mathcal{I}$ so that $y^{\mathcal{I}} = l$ for all variables $y \in e$ where l is some string constant of length $len^{\mathcal{I}}_{b_i}$ that we have not used so far. The existence of such a constant is guaranteed by rule Card, which makes sure that $len^{\mathcal{I}}_{b_i}$ is large enough.

Let $\mathcal{I}_K$ be the modified $\mathcal{I}$. To see that $\mathcal{I}_K$ is well-defined as an interpretation and, like $\mathcal{I}$, is a model of T_{SRL} it is enough to observe that, by construction,

$$\text{for all integer variables } z \text{ of } K,\; z^{\mathcal{I}_K} = z^{\mathcal{I}} \tag{3}$$

$$\text{for all string variables } x \text{ of } K,\; x^{\mathcal{I}_K} \text{ has length } (\mathsf{len}\, x)^{\mathcal{I}} \tag{4}$$

We show next that this model satisfies $S \cup A$. □

Lemma 7 *The model $\mathcal{I}_K$ satisfies A.*

Proof Immediate consequence of (3), (4) and the fact that the model $\mathcal{I}$ used to construct $\mathcal{I}_K$ satisfies A. □

To show that $\mathcal{I}_K$ satisfies S we need a few intermediate results.

Lemma 8 *Let $(K_i)_{i=1,\ldots,m} = (\langle S_i, A_i, R_i, F_i, N_i, C_i, B_i\rangle)_{i=1,\ldots,m}$ be the sequence of current configurations in the derivation of K, with $K = K_m$. Let $(\succ_i)_{i=1,\ldots,m}$ be the sequence of binary relations over $\mathcal{T}_{S_i}$ defined as follows:*

1. $\succ_0 := \emptyset$;
2. $\succ_{j+1} := \succ_j \cup \{(t, s)\} \setminus \{(s, u) \mid s \succ_j u\}$ *if K_{j+1} is derived from K_i by an application of* S-Cycle *where s, t are as in the premise of those rules;*
3. $\succ_{j+1} := \succ_j$, *otherwise.*

For each $i \in 1, \ldots, m$, the relation $\succ_i$ is well founded. Moreover, none of its minimal elements is in C_i.

Proof We prove both claims simultaneously by induction on i.

For the well foundedness one it is enough to show that the relations, which are finite, are acyclic. The base case and the inductive step when $\succ_{j+1} := \succ_j$ are trivial. For case (2), since $\succ_i$ is acyclic by induction hypothesis, $\succ_{i+1}$ has a cycle only if $s \succ^*_{i+1} t$. But this is impossible because (i) $s \neq t$ by the premises of S-Cycle, and (ii) s is a minimal element for $\succ_{i+1}$.

The term s is not in C_{i+1} because it is explicitly removed by S-Cycle from the C component of the current configuration if present. The other minimal elements of $\succ_{i+1}$, if any, are all minimal for $\succ_i$ by construction of $\succ_{i+1}$, so they are not in C_{i+1} by induction hypothesis. □

[7] Refer back to Fig. 7 for a definition of len_b.

[8] Observe that $N\, e$ is defined by point (ii) of Definition 6.

Lemma 9 *For all terms t of sort* Str *in $\mathcal{T}_S$, $t^{\mathcal{I}_K} = (\mathsf{con}\,(N\,[t]))^{\mathcal{I}_K}$.*

Proof Let $t \in \mathcal{T}_S$ of sort Str. We prove the claim by induction on $|t^{\mathcal{I}_K}|$. If t is a variable, the statement holds directly by our construction of $\mathcal{I}_K$, so suppose t is not a variable. By saturation of K and the premises of $\mathsf{N\text{-}Form1}$ and $\mathsf{N\text{-}Form2}$, then either $F\,t = N\,[t]$ or $t \in C$. We consider the two cases separately.

($F\,t = N\,[t]$) If t is a string constant l then $N\,[t] = (l)$ and the claim is immediate. If t is a term of the form $\mathsf{con}(t_1, \ldots, t_n)$ with $n > 1$, then, by construction of F and N, $N\,[t] = (\mathbf{s}_1, \ldots, \mathbf{s}_n)\!\downarrow$, where $\mathbf{s}_1 = N\,[t_1], \ldots, \mathbf{s}_n = N\,[t_n]$. For all $i \in \{1, \ldots, n\}$ we have that $|t_i^{\mathcal{I}_K}| < |t^{\mathcal{I}_K}|$ by Lemma 6([3.]) and Lemma 7; so, by induction hypothesis, $t_i^{\mathcal{I}_K} = (\mathsf{con}\,\mathbf{s}_i)^{\mathcal{I}_K}$. Thus, by definition of $\downarrow$ and construction of $\mathcal{I}_K$, $t^{\mathcal{I}_K} = (\mathsf{con}(t_1, \ldots, t_n))^{\mathcal{I}_K} = t_1^{\mathcal{I}_K} \cdots t_n^{\mathcal{I}_K} = (\mathsf{con}\,\mathbf{s}_1)^{\mathcal{I}_K} \ldots (\mathsf{con}\,\mathbf{s}_n)^{\mathcal{I}_K} = (\mathsf{con}\,(\mathbf{s}_1, \ldots, \mathbf{s}_n))^{\mathcal{I}_K} = (\mathsf{con}\,(\mathbf{s}_1, \ldots, \mathbf{s}_n)\downarrow)^{\mathcal{I}_K} = (\mathsf{con}\,(N\,[t]))^{\mathcal{I}_K}$.

($t \in C$) Let $\succ\, = \,\succ_m$ where $\succ_m$ is the well founded relation defined in Lemma 8. By that lemma, t is not a minimal for $\succ$. However, by the previous case and the lemma again, $s^{\mathcal{I}_K} = (\mathsf{con}\,(N\,[s]))^{\mathcal{I}_K}$ for all minimal terms s of $\succ$. We show below that $\bar{s} \approx \bar{t} \in \widehat{S}$ and $\bar{s}^{\mathcal{I}_K} = \bar{t}^{\mathcal{I}_K}$ for all $\bar{s}, \bar{t}$ such that $\bar{t} \succ \bar{s}$. From that it will then follow that $t^{\mathcal{I}_K} = (\mathsf{con}\,(N\,[t]))^{\mathcal{I}_K}$. Let $\bar{t} \succ \bar{s}$. By construction of $\succ$, $(\bar{t}, \bar{s})$ must have been added to it by an application of $\mathsf{S\text{-}Cycle}$, hence $\bar{t}$ must be of the form $\mathsf{con}\,(\mathbf{u}_1, \bar{s}, \mathbf{u}_2)$. Since constraints are never removed from the S component of the current configuration during a derivation, we also have that $u \approx \epsilon \in \widehat{S}$ for all u in $(\mathbf{u}_1, \mathbf{u}_2)$. Now, if $|\bar{t}^{\mathcal{I}_K}| = 0$, since $\mathcal{I}_K$ is a model of T_{SRL} and $\bar{t} = \mathsf{con}\,(\mathbf{u}_1, \bar{s}, \mathbf{u}_2)$, it must be that $u^{\mathcal{I}_K} = \epsilon$ for all u in $(\mathbf{u}_1, \mathbf{u}_2)$. If $|\bar{t}^{\mathcal{I}_K}| > 0$, let u in $(\mathbf{u}_1, \mathbf{u}_2)$. Since $|\epsilon| = 0$ and $u \in [\epsilon]$, we have by our inductive hypothesis that $u^{\mathcal{I}_K} = (\mathsf{con}\,(N\,[u]))^{\mathcal{I}_K} = (\mathsf{con}\,(N\,[\epsilon]))^{\mathcal{I}_K} = (\mathsf{con}(\epsilon))^{\mathcal{I}_K} = \epsilon$. In both cases, we can conclude that $\bar{s}^{\mathcal{I}_K} = \bar{t}^{\mathcal{I}_K}$ and $\bar{s} \approx \bar{t} \in \widehat{S}$. □

Lemma 10 *For all buckets $b \in B$ and distinct $[x], [y] \in b$, $(\mathsf{con}\,(N\,[x]))^{\mathcal{I}_K} \neq (\mathsf{con}\,(N\,[y]))^{\mathcal{I}_K}$.*

Proof We prove the claim by induction, assuming the property holds for all buckets b' where $len_{b'}^{\mathcal{I}_K} < len_b^{\mathcal{I}_K}$.

First observe that $|x^{\mathcal{I}_K}| = |y^{\mathcal{I}_K}| = len_b^{\mathcal{I}_K}$ by construction of $\mathcal{I}_K$. Given a derivation of configuration K, suppose $[x]$ was last added to b after $[y]$ at some step i in the derivation applied to a configuration K_i in the derivation of K. Then $[x]$ was necessarily added by an application of $\mathsf{D\text{-}Add}$ to some term $s \in [x]$. We claim that in K, the equivalence classes $[x]$ and $[y]$ have respectively the same normal form $(\mathbf{w}, u, \mathbf{u}_1)$ and $(\mathbf{w}, v, \mathbf{v}_1)$ they had in K_i. To see this note that, once computed, the normal form of an equivalence class is never modified by the calculus.[9] Since K is saturated, this implies that $N\,[x] = N_i\,[x]$ and $N\,[y] = N_i\,[y]$. Moreover, since $S_i \subseteq S$, we have by the premise of $\mathsf{D\text{-}Add}$ that $S \models \mathsf{len}\,u \approx \mathsf{len}\,v$ and $u \not\approx v \in \widehat{S}$. The disequality $u \not\approx v$ implies that u and v are distinct (atomic) terms, otherwise $\mathsf{S\text{-}Conflict}$ would apply to K. This means that $u^{\mathcal{I}_K}$ and $v^{\mathcal{I}_K}$ are distinct by construction. It also implies, by saturation of K, that $[u]$ and $[v]$ are in the same bucket b'.

Now, if $\mathbf{w}, \mathbf{u}_1, \mathbf{v}_1$ are all empty tuples, $(\mathsf{con}\,(N\,[x]))^{\mathcal{I}_K} = u^{\mathcal{I}_K} \neq v^{\mathcal{I}_K} = (\mathsf{con}\,(N\,[y]))^{\mathcal{I}_K}$. If, on the other hand, one of $\mathbf{w}, \mathbf{u}_1$ and $\mathbf{v}_1$ is non-empty, we have by Lemma 6 and by construction of $\mathcal{I}_K$ that $|u^{\mathcal{I}_K}| < len_b^{\mathcal{I}_K}$ or $|v^{\mathcal{I}_K}| < len_b^{\mathcal{I}_K}$. In either case, then $len_{b'}^{\mathcal{I}_K} < len_b^{\mathcal{I}_K}$. By induction hypothesis for bucket b', we then have that $(\mathsf{con}\,(N\,[u]))^{\mathcal{I}_K} \neq (\mathsf{con}\,(N\,[v]))^{\mathcal{I}_K}$ and thus $(\mathsf{con}\,(N\,[x]))^{\mathcal{I}_K} \neq (\mathsf{con}\,(N\,[y]))^{\mathcal{I}_K}$ as well. □

[9] It is at most recomputed from scratch after an application of Reset.

Lemma 11 *The model $\mathcal{I}_K$ satisfies S.*

Proof By Properties (2) and (4) of the model construction, $\mathcal{I}_K$ satisfies all constraints of the form $\mathsf{len}\, x \approx \mathsf{len}\, y$ or $\mathsf{len}\, x \not\approx \mathsf{len}\, y$ in S. So we only need to show that $\mathcal{I}_K$ satisfies all the equalities $s \approx t$ and disequalities $s \not\approx t$ in S between terms of sort Str.

For $s \approx t$, this is an immediate consequence of Lemma 9 after observing that $[s] = [t]$ and so $s^{\mathcal{I}_K} = (\mathsf{con}\,(N\,[s]))^{\mathcal{I}_K} = (\mathsf{con}\,(N\,[t]))^{\mathcal{I}_K} = t^{\mathcal{I}_K}$.

For $s \not\approx t$, let x, y be variables such that $s \in [x]$ and $t \in [y]$ and note that $[x]$ and $[y]$ are distinct as otherwise $\mathsf{S\text{-}Conflict}$ would apply to K. Since K is saturated, B is a partition of $\mathbf{E}_S$ so there are $b_1, b_2 \in B$ such that $[x] \in b_1$ and $[y] \in b_2$. If b_1 and b_2 are different buckets, by construction of $\mathcal{I}_K$ we have that $|x^{\mathcal{I}_K}| \neq |y^{\mathcal{I}_K}|$, which implies that $x^{\mathcal{I}_K} \neq y^{\mathcal{I}_K}$. By Lemma 9, $s^{\mathcal{I}_K} = x^{\mathcal{I}_K}$ and $t^{\mathcal{I}_K} = y^{\mathcal{I}_K}$, hence $s^{\mathcal{I}_K} \neq t^{\mathcal{I}_K}$. If b_1 and b_2 are the same, then $(\mathsf{con}\,(N\,[x]))^{\mathcal{I}_K} \neq (\mathsf{con}\,(N\,[y]))^{\mathcal{I}_K}$ by Lemma 10. Then $s^{\mathcal{I}_K} \neq t^{\mathcal{I}_K}$ follows from the fact that $s^{\mathcal{I}_K} \approx (\mathsf{con}\,(N\,[x]))^{\mathcal{I}_K}$ and $t^{\mathcal{I}_K} \approx (\mathsf{con}\,(N\,[y]))^{\mathcal{I}_K}$ by Lemma 9. □

Theorem 2 (Solution soundness) *If a derivation tree with initial configuration $\langle S_0, A_0, \emptyset, \emptyset, \emptyset, \emptyset, \emptyset\rangle$ has a saturated branch with no applications of* $\mathsf{F\text{-}Loop}$ *then the set $S_0 \cup A_0$ is satisfiable in T_{SRL}.*

Proof Assume there exists a derivation tree with root node $\langle S_0, A_0, \emptyset, \emptyset, \emptyset, \emptyset, \emptyset\rangle$ containing a saturated configuration $K = \langle S, A, R, F, N, C, B\rangle$ with $R = \emptyset$. By Lemmas 7 and 11, there is a T_{SRL}-model $\mathcal{I}_K$ that satisfies $S \cup A$. It is easy to show by structural induction on derivation trees that $S \cup A$ is a superset of $S_0 \cup A_0$. It follows that $S_0 \cup A_0$ is satisfiable in T_{SRL}. □

5 Proof procedure

A proof procedure for the calculus is a procedure that constructs a derivation from an initial configuration by applying the rules according to a certain application strategy. We present here an abstract, non-deterministic proof procedure that captures the main features of a concrete and more sophisticated one we have implemented in CVC4. The procedure is based on the repeated application of the derivation rules of the calculus according to the six steps below.

Proof procedure steps

1. *Reset* Apply Reset to reset buckets, and flat and normal forms.
2. *Check for conflicts* Apply $\mathsf{S\text{-}Conflict}$ or $\mathsf{A\text{-}Conflict}$ if possible.
3. *Exchange equalities* Propagate to completion any entailed equalities between S and A using $\mathsf{S\text{-}Prop}$ and $\mathsf{A\text{-}Prop}$.
4. *Add length constraints* Apply Len and then $\mathsf{Len\text{-}Split}$ to completion.
5. *Compute normal forms for equivalence classes* Apply $\mathsf{S\text{-}Cycle}$ to completion and then the rules in Fig. 5 to completion. If this does not produce a total map N, there must be some $s \approx t \in \widehat{\mathsf{S}}$ such that $\mathsf{F}\, s$ and $\mathsf{F}\, t$ have respectively the form $(\mathbf{w}, u, \mathbf{u}_1)$ and $(\mathbf{w}, v, \mathbf{v}_1)$ 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 $\mathsf{len}\, x \not\approx \mathsf{len}\, y$, apply $\mathsf{L\text{-}Split}$ to them; otherwise, apply any applicable rules from Fig. 6, giving preference to $\mathsf{F\text{-}Unify}$.
6. *Partition equivalence classes into buckets* First apply $\mathsf{D\text{-}Base}$ and $\mathsf{D\text{-}Add}$ 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. If for some bucket b, we have neither $S \models \mathsf{len}\, x \approx len_b$ nor $S \models \mathsf{len}\, x \not\approx$

len_b, then split on this using L-Split. Otherwise, we must have 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 $x \not\approx y \notin \widehat{\mathsf{S}}$, split on $x \approx y$ and $x \not\approx y$ using S-Split. Otherwise, let $[y] \in b$ such that $x \not\approx y \in \widehat{\mathsf{S}}$. 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 such as $u \in [x_u]$ and $v \in [x_v]$. 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 $\widehat{\mathsf{S}}$, 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.

7. *Add length constraint for cardinality* Apply Card to completion.

In addition to applying the steps above in the given order, the proof procedure also conforms to the following directives.

– Grow the derivation-tree depth first.
– When applying a branching rule try the left-branch configuration first.
– Interrupt a step and restart with Step 1 as soon as S changes.
– Keep cycling through the steps and stop as soon as a closed derivation tree or a saturated one is derived.

We illustrate the procedure's workings with a few examples. The first one shows how the procedure constructs a closed tree from an unsatisfiable input problem.

Example 4 Suppose we start with

$$\mathsf{A} = \emptyset \text{ and } \mathsf{S} = \{\mathsf{len}\, x \approx \mathsf{len}\, y,\ x \not\approx \epsilon,\ z \not\approx \epsilon,\ \mathsf{con}(x, l_1, z) \approx \mathsf{con}(y, l_2, z)\}$$

where l_1, l_2 are distinct constants of the same positive length and x, y and z are string variables. After checking for conflicts, the procedure applies A-Prop, S-prop, 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 it uses 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. □

The next example shows how the procedure derives a saturated configuration from a satisfiable input problem.

Example 5 Suppose now the input constraints are

$$\mathsf{A} = \emptyset \text{ and } \mathsf{S} = \{\mathsf{len}\, x \approx \mathsf{len}\, y,\ x \not\approx \epsilon,\ z \not\approx \epsilon,\ \mathsf{con}(x, l_1, z) \not\approx \mathsf{con}(y, l_2, z)\}$$

with l_1, l_2, x, y and z as in Example 4. 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 $\mathsf{D\text{-}Base}$, then neither $\mathsf{D\text{-}Base}$ (since $S \models \mathsf{len}\, x \approx \mathsf{len}\, y$) nor $\mathsf{D\text{-}Add}$ (since $x \not\approx y \notin \widehat{\mathsf{S}}$) is applicable to $[y]$. So it applies $\mathsf{S\text{-}Split}$ to x and y. In the branch where $x \approx y$, the proof procedure subsequently restarts, computing 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 $\mathsf{D\text{-}Add}$, 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. □

The final example shows that the procedure is not terminating in general. It diverges when trying to establish the unsatisfiability of some inputs where the same variable appears on both sides of an equality.

Example 6 Suppose the input constraints are

$$\mathsf{A} = \emptyset \text{ and } \mathsf{S} = \{\mathsf{len}\, x \approx 2 * n,\ \mathsf{con}(a, b, x) \approx \mathsf{con}(x, b, a)\}$$

where a and b are distinct constants of length 1 and x and n are variables. Note that this input is unsatisfiable, since all models of the second equality interpret x as a word of the form $(ab)^m a$ with $m \geq 0$, which has odd length, in contradiction with the constraint expressed by the first equality. The procedure applies Len and $\mathsf{Len\text{-}Split}$ to completion, after which it will determine all non-conflict configurations are such that $\mathsf{len}\, x > 0$. Using the rules in Fig. 5, it will construct flat forms (a, b, x) and (x, b, a) for the terms $\mathsf{con}(a, b, x)$ and $\mathsf{con}(x, b, a)$. Since $\mathsf{len}\, x \approx 2*n$, we know that $\mathsf{len}\, x \not\approx 1 \approx \mathsf{len}\, a$, and hence after applying $\mathsf{L\text{-}Split}$ to the equality $\mathsf{len}\, x \approx \mathsf{len}\, a$, the rule $\mathsf{F\text{-}Loop}$ will apply to the flat forms (a, b, x) and (x, b, a) in all branches when $\mathsf{A\text{-}Conflict}$ does not apply. As a result, it will add constraints $x \approx \mathsf{con}(z_2, z)$, $\mathsf{con}(a, b) \approx \mathsf{con}(z_2, z_1)$, and $\mathsf{con}(b, a) \approx \mathsf{con}(z_1, z_2)$ to S and z in $\mathsf{star}(\mathsf{set}\ \mathsf{con}(z_1, z_2))$ to R. After applying Len to completion and $\mathsf{Len\text{-}Split}$ on z_1 and z_2, it can be shown that all non-conflicting child configurations in the derivation tree that are saturated with respect to the rules in Fig. 6 are such that $z_1 \approx b$ and $z_2 \approx a$. Since Len is applied to $x \approx \mathsf{con}(z_2, z)$, we have that $\mathsf{len}\, x \approx \mathsf{len}\, z_2 + \mathsf{len}\, z$. Since $\mathsf{len}\, z_2 \approx 1$ and $\mathsf{len}\, x \approx 2 * n$, this implies that $\mathsf{len}\, z$ is non-zero. Thus, running the proof strategy described in this section concludes with a configuration in which the only applicable rule of the calculus in this state is $\mathsf{R\text{-}Star}$. This rule does not suffice to show this example is unsatisfiable: applying $\mathsf{R\text{-}Star}$ to z in $\mathsf{star}(\mathsf{set}\ \mathsf{con}(z_1, z_2))$ will add $z \approx \mathsf{con}(z_1, z_2, z')$ to S and z' in $\mathsf{star}(\mathsf{set}\ \mathsf{con}(z_1, z_2))$ to R for some z'. After applying Len to $\mathsf{con}(z_1, z_2, z')$ we will similarly determine that the length of z' must be non-zero, and repeated applications of $\mathsf{R\text{-}Star}$ could continue in this way indefinitely. □

Although we will not do it here, we can show that iterative-deepening variants of the proof procedure given here are *solution complete*, that is, always able to generate a saturated derivation tree when the input problem is satisfiable in T_{SRL}. One simple and practical way to do that is to bound the sum of the lengths of the string variables in the original problem, and then attempt to solve the bounded version for increasing values of the bound.

In contrast, we do not have a proof currently that our calculus admits proof procedures that are *refutation complete*, that is, guaranteed to generate a refutation for any unsatisfiable input problem. The main issue is termination. Specifically, we currently do not have a proof that our calculus admits proof-procedures that always terminate with a saturated or closed derivation tree. The main problem is represented by *non-linear* word equations, i.e., positive string constraints containing more than one occurrence of the same variable. The calculus does mitigate the risk of non-termination introduced by such constraints by means of the $\mathsf{F\text{-}Loop}$ rule which is able to break certain derivation loops, as shown in the example below.

Example 7 Suppose the input constraint is

$$\mathsf{A} = \emptyset \text{ and } \mathsf{S} = \{\mathsf{con}(x, a) \approx \mathsf{con}(b, x)\}$$

where x is a string variable and a and b are distinct string constants of length 1. It is possible to show by a simple inductive argument that this problem is unsatisfiable. Suppose we modify the proof procedure so that it never applies $\mathsf{F\text{-}Loop}$. At some point $\mathsf{F\text{-}Split}$ is applicable. On one branch (the other branch is similar), the rule adds $x \approx \mathsf{con}(b, x')$ to S with x' a fresh variable. By normalization, we eventually have a configuration with $\mathsf{F}\,\mathsf{con}(x, a) = (b, x', a)$, $\mathsf{F}\,\mathsf{con}(b, x) = (b, b, x')$ and $\mathsf{S} \models \mathsf{len}\,x' \not\approx \mathsf{len}\,b$. Rule $\mathsf{F\text{-}Split}$ is applied again, adding $x' = \mathsf{con}(b, x'')$ to S with x'' fresh. This sort of splitting is triggered indefinitely , each time with a fresh variant of $x \approx \mathsf{con}(b, x')$, making the procedure diverge.

In contrast, if $\mathsf{F\text{-}Loop}$ is applied on this example, two equalities like $a \approx \mathsf{con}(z_1, z_2)$ and $b \approx \mathsf{con}(z_2, z_1)$ with z_1 and z_2 fresh variables are added to S. Those two equalities allow the procedure to generate a closed tree eventually. □

Unfortunately, $\mathsf{F\text{-}Loop}$ is not enough to guarantee termination in all cases. Our current proof procedure will diverge on certain problems involving non-linear word equations and additional length constraints on their variables, as illustrated by Example 6. Furthermore, the rule introduces membership constraints even in problems that originally had none. More importantly, these constraints contain *symbolic regular expressions*, that is, regular expressions with occurrences of string variables. Such expressions make it arduous to prove termination because they can actually denote non-regular languages in general.

Abdulla et al. [1] describe a decision procedure for a subset of the language of T_{SRL}-constraints that satisfies certain acyclicity properties. We conjecture that our proof-procedure is terminating on that language and hence is an alternative decision procedure for it. A proof of this conjecture, however, is non-trivial and out of the scope of this work.

6 Implementation

We have described a calculus and a proof procedure based on it that is solution and refutation sound for finite sets of T_{SRL}-constraints. In this section, we describe how an efficient solver based on this proof procedure can be integrated into an DPLL(T)-based SMT solver.

The DPLL(T) framework used by modern SMT solvers 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 formulas (specifically, clauses) and a set M of literals representing a partial Boolean assignment for the atoms of 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 in the current configuration $\langle \mathsf{S}, \mathsf{A}, \mathsf{R}, \mathsf{F}, \mathsf{N}, \mathsf{C}, \mathsf{B} \rangle$), arithmetic constraints (the set A) and regular language constraints (the set R). These sets are subsequently given to two independent solvers, the *string solver* and the *arithmetic solver*. The rules $\mathsf{A\text{-}Prop}$ and $\mathsf{S\text{-}Prop}$ model the standard mechanism for Nelson–Oppen theory combination, where entailed equalities between shared terms are exchanged between these solvers. The satisfiability check performed by the arithmetic solver is modeled by the rule $\mathsf{A\text{-}Conflict}$. 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 case splitting done by the string solver (with rules $\mathsf{S\text{-}Split}$ and $\mathsf{L\text{-}Split}$) is achieved by means of the *splitting on demand* paradigm [2], 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 $\mathsf{F\text{-}Split}$ and $\mathsf{D\text{-}Split}$ can be implemented by adding a lemma of the form $\psi \Rightarrow (\varphi_1 \vee \varphi_2)$ to F, where φ_1 and φ_2 are new literals. For instance, in the case of $\mathsf{F\text{-}Split}$, 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, $\mathsf{Len\text{-}Split}$, and Card involve adding constraints to A. This is done by the string solver by adding to F lemmas containing arithmetic constraints. For instance, if $x \approx \mathsf{con}(y, z) \in \widehat{\mathsf{S}}$, 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 $\widehat{\mathsf{S}}$ 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 $\mathsf{S\text{-}Cycle}$, $\mathsf{F\text{-}Unify}$ or standard congruence closure rules. The explanation $\psi_{s,t}$ is a conjunction of string constraints in M such that $\psi_{s,t} \models_{\mathsf{SRL}} s \approx t$. For $\mathsf{F\text{-}Unify}$, the string solver maintains an explanation ψ for the flat form of each term $t \in \mathcal{D}(\mathsf{F})$ where $\psi \models_{\mathsf{SRL}} t \approx \mathsf{con}(\mathsf{F}\, t)$. When a configuration is determined to be unsatisfiable by $\mathsf{S\text{-}Conflict}$ (which in practice happens when $s \approx t, s \not\approx t \in \widehat{\mathsf{S}}$ for some s, t), the solver replaces the occurrence of $s \approx t$ with its corresponding explanation ψ, and then replaces the equalities in ψ with their corresponding explanations, and so on, until ψ consists of only equalities from M. Then it reports $\psi \Rightarrow s \approx t$ as a conflict clause.

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

7 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.[10] 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 three of the string solvers mentioned in Sect. 1.1: KALUZA (latest version from its website), S3 (latest version from its website), and Z3- STR (version 20140720). 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. We also considered an experimental version of NORN supporting the concrete syntax of our benchmarks. We were, however, unable to produce meaningful results with that version.[11] As a consequence, we will not discuss NORN in the following.

All results in this section were produced on StarExec [28], a specialized public execution service that allows researchers to run comparative evaluations of logical solvers. The service

[10] CVC4 is publicly available at http://cvc4.cs.nyu.edu/.

[11] For many benchmarks, NORN, which runs on the Java virtual machine, crashed when executed on StarExec because of insufficient resources in the JVM. Also, for satisfiable problems, NORN returns solutions in a non-standard format, which made it difficult for us to validate those models.

has been hosting regular solver competitions (e.g., SMT-COMP) since 2013. All experiments were run in the *all.q* queue of the StarExec cluster, which consists of 160 identical nodes. Each node uses a 2.40 GHz Intel Xeon E5-2609 with 10 MB cache and 256 GB main memory.[12]

Modulo superficial differences in the concrete input syntax, CVC4, KALUZA, S3, and Z3-STR accept as input a set of T_{SRL} constraints, with the exception that Z3- STR does not accept regular language constraints. All tools report on the satisfiability of the input problem with a sat, unsat or unknown answer. In the first case, CVC4 and S3 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 beforehand in the input file.

An initial series of regression tests on all four tools revealed several usability and correctness issues with KALUZA and a few with S3 and Z3- STR. In KALUZA, they were caused by bugs in its top level script which communicates with different external tools, e.g., the solvers Yices and Hampi, via the file system. Those bugs range from failure to clean up temporary files to an incorrect use of the Unix grep tool to extract information from the output of underlying 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.

Our previous experiments [18] found that in several cases Z3- STR returned *spurious solutions*, i.e., assignments to the input variables that do not in fact satisfy the input problem. Also, it classified some satisfiable problems as unsat. Prompted by our inquiries, the Z3- STR developers produced and shared with us a new version of Z3- STR that fixes the spurious solutions problem (Version 20140720). We are using that version for the comparison in this paper. In that version of Z3- STR, we have detected no incorrect solutions on these Kudzu benchmarks. However, we have discovered that Z3- STR sometimes generates incomplete solutions, that is, it generates a satisfying assignment only for a subset of the variables in the input benchmark. This happened for a large amount of benchmarks (6404) in our set. For example, for the benchmark `sat/big/100.corecstrs.readable.smt2`, the returned solution does not contain an assignment for three of the 243 declared input variables.

Similar errors were observed with S3, which is an extension for Z3- STR, although this solver returns a smaller number of incomplete solutions (4826) for the same set of benchmarks than Z3- STR. In contrast, S3 reports 95 unsound answers, i.e., it reports unsat in 95 cases where both CVC4 and Z3- STR find a verifiable solution.

On our full set of benchmarks, we did not find any evidence of erroneous behavior in CVC4 when compared with the other three solvers. Every solution produced by CVC4 was *confirmed* by CVC4, S3 and Z3- STR by appending 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 either S3 or Z3- STR.

7.1 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

[12] Both the solvers and the results are available, by logging in as a guest user, at https://www.starexec.org/starexec/secure/details/job.jsp?id=6875 (CVC4) and https://www.starexec.org/starexec/secure/details/job.jsp?id=6891 (Z3- STR and S3).

Table 1 Comparative results

Result	CVC4	S3			Z3-str		Kaluza		Kaluza-orig	
	✓	×	?	✓	?	✓	×	✓	×	✓
Sat	33,772	0	4826	26,099	6404	26,403	n/a	25,468	n/a	3
Unsat	11,625	95		11,313		11,340	7154	13,435	27,450	805
Timeout	1887			1647		2773		84		84
Unknown	0			896		364		3		0
Error	0			2414		0		1140		18,942
Avg. mem	14,130			60,447		82,160		n/a		n/a

on the KALUZA website [26]. The discarded problems either had syntax errors or included a macro function (CapturedBrack) whose meaning is not fully documented.[13] We translated those benchmarks into CVC4's extension of the SMT-LIB 2 format to the language of T_{SRL}[14] and into the Z3- STR format. Some benchmarks contain regular membership constraints (s 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, S3, Z3- STR and two versions of KALUZA, the original one and the one with our debugged script, on each benchmark with a 2-minute CPU time limit. (We obtained the same results with timeouts up to 15 minutes.) The results are summarized in Table 1. 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 (up to) three columns: the × column contains the number of provably incorrect answers, the ? column contains the number of incomplete solutions, and the ✓ column contains the rest. By *provably incorrect* here we mean a spurious solution or an unsat answer for a problem that actually has a verified 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.

All of CVC4's 33,772 sat answers were corroborated by a confirmed solution. In addition, any problem that is classified as sat by either S3 or Z3- STR, can be solved by CVC4. Among these 33,772 sat answers, S3 agreed on 30,925 and generated 26,099 verifiable full solutions but returned 4826 incomplete solutions; Z3- STR agreed on 32,807 and generated 26,403 verifiable full solutions but returned 6404 incomplete solutions. Neither S3 or Z3- STR returned a spurious solution.

None of the 11,625 unsat answers provided by CVC4 were provably incorrect, meaning that no other solver generated a confirmed solution for these benchmarks. Among these 11,625 unsat answers, both S3 and Z3- STR agreed on 11,313, and timed out on the remaining 312.

[13] The Kaluza documentation does not specify the meaning of the function when its second argument, an integer, is greater than 0.

[14] 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.

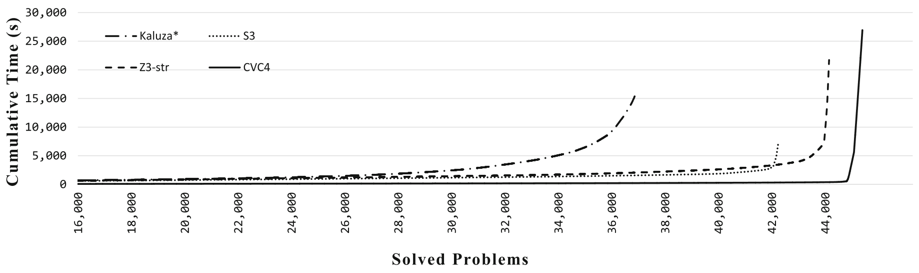

Fig. 8 Runtime comparison of CVC4, S3, Z3- STR and the amended KALUZA. Times are in seconds. The 0 line stands for benchmarks cumulatively solved within 1 s

With respect to the total unsat answers on all benchmarks, S3 erroneously classified 95 sat problems as unsatisfiable, while Z3- STR distinctively reports an additional 27 unsat problems. KALUZA reported 11,394 unsat problems and 25,468 sat problems (unverifiable because of the absence of solutions), erroneously classified 7154 as unsatisfiable, reported unknown for 3, produced an error for 562, and timed out on 84.

CVC4 timed out on 1887 problems, but produced no errors and no unknown answers. Z3- STR timed out on 2773 problems, and produced 364 unknown answers. S3 timed out on 1647 problems and produced 896 unknown answers and 2414 errors.

These results provide strong evidence that CVC4's string solver is sound, i.e., any unsat answers from CVC4 can be trusted. They also show that CVC4's string solver answers sat more often than S3 and Z3- STR and KALUZA, providing a correct solution in each case. Thus, it is overall the best tool for both satisfiable and unsatisfiable problems.

In terms of run time performance, a comparison with KALUZA is not very meaningful because of its high unreliability and the unverifiability of its sat answers. However, an analysis of our detailed results shows that CVC4 has nonetheless better runtime performance overall with respect to all of the solvers. This can be easily seen from the cactus plot in Fig. 8, which shows for each of the four systems how many benchmarks it cumulatively solves within a certain amount of time.

Thanks to the StarExec infrastructure, we can accurately measure the memory consumption of each solver on every benchmark. CVC4 consumed the least memory on average (14 MB), compared to S3 (60 MB) and Z3- STR (82 MB).

8 Conclusion and further work

We have presented an 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 described the approach abstractly as a calculus and a general proof procedure for that calculus. We have proven that the proof procedure is both solution sound and refutation sound. We have also given experimental evidence that our implementation in the SMT solver CVC4 is highly competitive with existing tools.

We are currently extending the scope of our string solver to support a richer language of string constraints that occur often in practice, especially in security 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 efficiently by extending the calculus described in this paper. We are also working on a more sophisticated approach for dealing with regular language constraints, using a separate dedicated solver that is similarly integrated into the DPLL(T) framework.

At the theoretical level, we would like to identify further interesting fragments of the T_{SRL} theory over which our proof procedure is both terminating and refutation complete.

Acknowledgments We thank the developers of Z3- STR for their technical support in using their tool and several clarifications on it, as well as for their prompt response to our inquiries. We also express our gratitude to the developers of the StarExec service for their assistance and for implementing additional features we requested while running our experimental evaluation on the service. Finally, we thank the anonymous reviewers for their supportive comments and for their valuable suggestions on improving the paper's presentation. The work described here was partially funded by NSF Grants #1228765 and #1228768. The second author was also supported in part by the European Research Council (ERC) Project Implicit Programming.

References

1. Abdulla PA, Atig MF, Chen YF, Holik L, Rezine A, Rummer P, Stenman J (2014) String constraints for verification. In: Biere A, Bloem R (eds) Proceedings of the 26th international conference on computer aided verification. Lecture notes in computer science, vol. 8559. Springer, Berlin
2. Barrett C, Nieuwenhuis R, Oliveras A, Tinelli C (2006) Splitting on demand in SAT modulo theories. In: Proceedings of LPAR'06. Lecture notes in computer science, vol. 4246. Springer, Berlin, pp 512–526
3. Barrett C, Sebastiani R, Seshia S, Tinelli C (2009) Satisfiability modulo theories. In: Biere A, Heule MJH, van Maaren H, Walsh T (eds) Handbook of satisfiability, vol 185, chap 26. IOS Press, Amsterdam, pp 825–885
4. Bjørner N, Tillmann N, Voronkov A (2009) 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. Lecture notes in computer science. Springer, pp 307–321
5. Brumley D, Caballero J, Liang Z, Newsome J (2007) Towards automatic discovery of deviations in binary implementations with applications to error detection and fingerprint generation. In: Proceedings of the 16th USENIX security symposium, Boston, MA, USA, 6–10 August 2007
6. Brumley D, Wang H, Jha S, Song DX (2007) Creating vulnerability signatures using weakest preconditions. In: 20th IEEE computer security foundations symposium, CSF 2007, 6–8 July 2007, Venice, Italy, pp 311–325
7. Christensen AS, Møller A, Schwartzbach MI (2003) Precise analysis of string expressions. In: Proceedings of the 10th international conference on static analysis. Lecture notes in computer science. Springer, Berlin, pp 1–18
8. De Moura L, Bjørner N (2008) Z3: an efficient SMT solver. In: Proceedings of the theory and practice of software, 14th international conference on tools and algorithms for the construction and analysis of systems. Lecture notes in computer science. Springer, Berlin, pp 337–340
9. Egele M, Kruegel C, Kirda E, Yin H, Song D (2007) Dynamic spyware analysis. In: 2007 USENIX annual technical conference on proceedings of the USENIX annual technical conference, ATC'07. USENIX Association, Berkeley, CA, USA, pp 18:1–18:14
10. Fu X, Li C (2010) 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, Skokie
11. Ganesh V, Minnes M, Solar-Lezama A, Rinard M (2013) Word equations with length constraints: what's decidable? In: Proceedings of the 8th international conference on hardware and software: verification and testing, HVC'12. Springer, Berlin, pp 209–226
12. Ghosh I, Shafiei N, Li G, Chiang W (2013) JST: an automatic test generation tool for industrial Java applications with strings. In: Proceedings of the 2013 international conference on software engineering, ICSE'13. IEEE Press, Piscataway, pp. 992–1001
13. Hooimeijer P, Veanes M (2011) An evaluation of automata algorithms for string analysis. In: Proceedings of the 12th international conference on verification, model checking, and abstract interpretation. Springer, Berlin, pp 248–262
14. Hooimeijer P, Weimer W (2009) A decision procedure for subset constraints over regular languages. In: Proceedings of the 2009 ACM SIGPLAN conference on programming language design and implementation. ACM, Dublin, pp 188–198

15. Hooimeijer P, Weimer W (2010) Solving string constraints lazily. In: Proceedings of the IEEE/ACM international conference on automated software engineering. ACM, New York, pp 377–386
16. Kiezun A, Ganesh V, Guo PJ, Hooimeijer P, Ernst MD (2009) HAMPI: a solver for string constraints. In: Proceedings of the eighteenth international symposium on Software testing and analysis. ACM, New York, pp 105–116
17. Li G, Ghosh I (2013) PASS: string solving with parameterized array and interval automaton. In: Bertacco V, Legay A (eds) Hardware and software: verification and testing. Lecture notes in computer science, vol 8244. Springer, Berlin, pp 15–31
18. Liang T, Reynolds A, Tinelli C, Barrett C, Deters M (2014) A DPLL(T) theory solver for a theory of strings and regular expressions. In: Biere A, Bloem R (eds) Proceedings of the 26th international conference on computer aided verification. Lecture notes in computer science, vol 8559. Springer, Berlin
19. Liang T, Tsiskaridze N, Reynolds A, Tinelli C, Barrett C (2015) A decision procedure for regular membership and length constraints over unbounded strings. In: Frontiers of combining systems. Springer, Berlin, pp 135–150
20. Makanin GS (1977) The problem of solvability of equations in a free semigroup. English transl. in Math USSR Sbornik, vol 32, pp 147–236
21. Namjoshi KS, Narlikar GJ (2010) Robust and fast pattern matching for intrusion detection. In: INFOCOM 2010. 29th IEEE international conference on computer communications, joint conference of the IEEE computer and communications societies, 15–19 March 2010, San Diego, CA, USA, pp 740–748
22. Nelson G, Oppen DC (1979) Simplification by cooperating decision procedures. ACM Trans Program Lang Syst 1(2):245–257
23. Nieuwenhuis R, Oliveras A, Tinelli C (2006) Solving SAT and SAT Modulo theories: from an abstract Davis–Putnam–Logemann–Loveland procedure to DPLL(T). J ACM 53(6):937–977
24. Perrin D (1989) Equations in words. In: Ait-Kaci H, Nivat M (eds) Resolution of equations in algebraic structures, vol 2. Academic Press, Cambridge, pp 275–298
25. Plandowski W (2004) Satisfiability of word equations with constants is in PSPACE. J ACM 51(3):483–496
26. Saxena P, Akhawe D (2010) Kaluza web site. http://webblaze.cs.berkeley.edu/2010/kaluza/
27. Saxena P, Akhawe D, Hanna S, Mao F, McCamant S, Song D (2010) A symbolic execution framework for JavaScript. In: Proceedings of the 2010 IEEE symposium on security and privacy. IEEE Computer Society, pp 513–528
28. Stump A, Sutcliffe G, Tinelli C (2014) Starexec: a cross-community infrastructure for logic solving. In: Demri S, Kapur D, Weidenbach C (eds) Proceedings of the 7th international joint conference on automated reasoning. Lecture notes in artificial intelligence. Springer, Berlin
29. Tillmann N, Halleux J (2008) Pex—white box test generation for .NET. In Beckert B, Hähnle R (eds) Tests and proofs. Lecture notes in computer science, vol 4966. Springer, Berlin, pp 134–153
30. Tinelli C, Harandi MT (1996) A new correctness proof of the Nelson–Oppen combination procedure. In: Baader F, Schulz KU (eds) Frontiers of combining systems: proceedings of the 1st international workshop (Munich, Germany). Applied logic. Kluwer Academic Publishers, Dordrecht, pp 103–120
31. Trinh MT, Chu DH, Jaffar J (2014) S3: a symbolic string solver for vulnerability detection in web applications. In: Yung M, Li N (eds) Proceedings of the 21st ACM conference on computer and communications security. ACM, New York, pp 1232–1243
32. Veanes M (2013) Applications of symbolic finite automata. In: Proceedings of the 18th international conference on implementation and application of automata, CIAA'13. Springer, Berlin, pp 16–23
33. Veanes M, Bjørner N, De Moura L (2010) Symbolic automata constraint solving. In: Proceedings of the 17th international conference on logic for programming, artificial intelligence, and reasoning. Lecture notes in computer science. Springer, Berlin, pp 640–654
34. Yu F, Alkhalaf M, Bultan T (2010) Stranger: an automata-based string analysis tool for php. In: Esparza J, Majumdar R (eds) Tools and algorithms for the construction and analysis of systems. Lecture notes in computer science, vol 6015. Springer, Berlin, pp 154–157
35. Zheng Y, Zhang X, Ganesh V (2013) Z3-str: a Z3-based string solver for web application analysis. In: Proceedings of the 2013 9th joint meeting on foundations of software engineering, ESEC/FSE 2013. ACM, New York, pp 114–124