SMT for Strings

Seminar: Satisfiability Checking

SS 2015

Meshkatul Anwer
Supervision: Cornelius Aschermann

July 3, 2015

Abstract

Our modern society relies on software systems and on-line services. Most of the times these systems are dealing with our private and sensitive data. Privacy violation is one of the major concerns about such web-based applications and on-line services. To provide a more secure environment for the Internet services, these web-based applications should be tested for vulnerability. The verification procedure and security analysis of such system rely on automatic solvers, which can check the satisfiability of constraints over a rich set of data types, including character strings. String is considered as the main information carrier in web-based communication.However, most traditional mathematical methods focus more on numbers and most string solvers today are standalone tools that can reason only about some fragment of the theory of strings and regular expressions. In this paper we try to explore a string solver for the theory of unbounded strings with concatenation, length, and membership in regular languages [1]. This theory is incorporated into the SMT solver cvc4 [2]. We will look into the detail of the decision procedure and how it finds a solution to formula containing strings.

1 Introduction

Stability and reliability of software systems has been a key issues in the protection of privacy and security. As we want more and more new features and functionalists into such systems, the complexity of the system is exploding. With the rise of web-based applications and online services, the amount of information sharing via networking has increased rapidly. As a result, the security of web applications is getting more and more attention. As the complexity of such systems is growing, manual analysis is getting harder and sometime impossible. In the last few years, a number of techniques which were originally developed for automatic verification purposes have been adopted for the software security analyses.

The ability to reason about string values is a major task in the field of security analyses. Especially in web-based applications, where the program inputs are often provided as strings. These strings are usually get processed through operations such as matching against regular expressions, concatenation, and substring extraction or replacement. We need to be able to formally reason about strings as well as other data types. In this paper, we will see how the automatic reasoning engines such as Satisfiability Module Theories (SMT) solvers, are helping to check the satisfiability of constraints over rich set of data types including strings. We will describe the SMT theory of strings [1] which is is incorporated into the SMT solver cvc4 [2].

2 Motivating Example

We are interested in SMT solver which helps to check the satisfiability of constraints over rich set of data types. Here we will see few examples such constraints on strings. Example 3 shows how we can express regular expression membership constraint.

Example 1: Find an assignment for x, where $x."ab" = "ba".x \wedge len(x) = 7$.

Example 2: Find a model for x, y and z, where $x."ab".y = y."ba".z \wedge z = x.y \wedge x."a" \neq "a".x$.

Example 3: Find a model for x and y, where both x and y are in the $RegEx(a * b)*$ and they are different but have the same length.

Where x, y and z are variables of type string, len is a function which returns the length of the string variable. Figure 1 and 2 show the encoding of Example 1 , Example 2 and Example 3 in smt-lib [3] format. The examples and code snippets are taken from [4].

```
...                                                  ...
(assert (= (str.++ x "ab") (str.++ "ba" x)))         (assert (= (str.++ x "ab" y) (str.++ y "ba" z)))
(assert (= (str.len x) 7))                           (assert (= z (str.++ x y)))
                                                     (assert (not (= (str.++ x "a") (str.++ "a" x))))
...                                                  ...
```

Figure 1: Encoding of Example 1 and Example 2 in smt-lib [3] format.

```
...
(assert(str.in.re x(re.* (re.++ (re.* (str.to.re "a") ) (str.to.re "b") ))))
(assert (str.in.re y(re.* (re.++ (re.* (str.to.re "a") ) (str.to.re "b") ))))

(assert (not (= x y)))
(assert (= (str.len x) (str.len y)))
...
```

Figure 2: Encoding of Example 3 in smt-lib [3] format.

In the context of security analysis, modern SMT solver is used as the core constraint solver. The idea is to reduce security problems to constraint satisfaction problems in some formal logic. If the constraints are unsatisfiable, the source code is free of any exploit; otherwise, there is an assignment (of variables in the constraints) that satisfies these constraints and defines a possible attack. Traditionally analysis tools use their own built-in constraint solvers. However, it is possible to encode a security analysis problem into a Satisfiability Modulo Theories (SMT) problem. Then the preexisting standard SMT solver, which combines a SAT solver with multiple specialized theory solvers can be used. As string is the dominant data type in modern web-applications, constraints over strings along with other data types need to be checked. In this paper we will have a look into a string solver.

3 Related work

The satisfiability problem of any reasonably comprehensive theory of character strings is undecidable [5]. However, a more restricted, but still quite useful, theories of strings is decidable. For example, any theories of fixed-length strings and some fragments over unbounded strings (e.g., word equations [6]). There has been a lot of research to identify the decidable fragments of this theory. And also a lot of efforts on the development of efficient solvers [7]. Most of these solvers can reason only about (some fragment of) the theory of strings and regular expressions. They also have strong restrictions on the expressiveness of their input language. These solvers works by reducing the problems to other data types, such as bit vectors.

4 Preliminaries

In this section, we will describe the preliminary concepts. Later we will present a formal description of the theory solver as a set of normalization rewrite rules, derivation rules and a proof procedure.

4.1 The DPLL(T) Procedure

In propositional logic, a formula is constructed from a set of Boolean variables using a set of logical connectives, such as $\wedge, \vee, \neg$. Boolean variables can be assigned a (truth) value that is either *true* or $false$. Given a Boolean formula, the Boolean satisfiability (SAT) problem is to answer whether we can find an assignment for those variables, such that the formula is evaluated to be true under the assignment. There are many decision procedures to solve the classic SAT problem. Among them, the DPLL procedure is the most frequently used in most modern SAT solvers.

The DPLL procedure takes a set of clauses as input. It returns *sat* if a logically consistent assignment can be found; otherwise, it returns *unsat*. During its computation, the procedure maintains an internal stack of literals (possibly with decision marks) to represent a partial assignment.

Whenever every clause is evaluated to be true under the current assignment, the procedure stops and returns *sat*. During a standard processing loop, the DPLL procedure processes the clause set in three steps:

1 It first checks whether the current partial assignment is consistent with the clause set by evaluation. If one of the clauses is evaluated to false and the stack contains at least one decision literal, the procedure pops out all literals in the stack till the nearest decision literal, then flips the sign of that decision literal and turns it into a propagation literal. This step is often referred to *Backtrack*. If one of the clauses is evaluated to $false$ and the stack does not contain one decision literal, the procedure stops and returns *unsat*.

2 After the inconsistent check, the procedure tries to push new propagation literals into the stack by logical deduction, e.g., Unit Propagation

3 If there are still unassigned variables, the procedure picks one heuristically, guesses its sign, and pushes it into the stack as a decision literal.

The procedure continues until all variables are assigned. In our context, the SAT solver should be able to handle incremental clause assertions.

4.2 The Nelson-Oppen Combination

In Boolean formulas, the signature only contains propositional variables and logical connectives; whereas in SMT formulas, the signature is extended to a set of predicate symbols, a set of function symbols and a set of non-Boolean variables. Those extended symbols can be interpreted in some background theories. An SMT formula is satisfiable if there exists a model that satisfies both the logical formula and the background theories.

Reasoning about an SMT formula usually involves several reasoning over several theories. We refer the procedure to reason over a theory T as a T-solver. Note that a T-solver can only handle conjunctions of literals. Given a formula over several theories where each theory has a T-solver, the Nelson-Oppen combination provides a procedure to reason about this constraint.

Here we present a brief describe of the deterministic Nelson-Oppen combination procedure. Let S_1 and S_2 are literals for T_1-solver and T_2-solver, respectively. Initially, constraints are partitioned into S_1 and S_2 based on their signature. Whenever S_1 is T_1-unsatisfiable or S_2 is T_2-unsatisfiable, the original problem is unsatisfiable. If T_1-solver derives a new T_2-equality from S_1 but not in S_2 yet, the new constraint will be put in S_2; similarly, if T_2-solver derives a new T_1-equality from S_2 but not in S_1 yet, the new constraint will be put in S_1. If no new equality can be derived from either T-solver and both S_1 and S_2 are T-satisfiable, the original problem is satisfiable. However, this procedure only works when every theory is convex. For example, integer linear arithmetics is not convex with respect to inequality.

Here we describe the non-deterministic Nelson-Oppen algorithm. Let C be a set of constants. An $arrangement A$ over C is a set of (dis-)equalities, such that for every pair of elements in C, either an equality or a dis-equality (between two) is in A. Now the algorithm work as follows: initially, constraints are partitioned into S_1 and S_2 based on their signature; for every arrangement A over the shared constants, if both $S_1 \cup A$ and $S_2 \cup A$ is T-satisfiable, the original problem is satisfiable; otherwise, it is unsatisfiable. Latter we will see how theory solvers communicate (dis-)equalities over shared terms with an example.

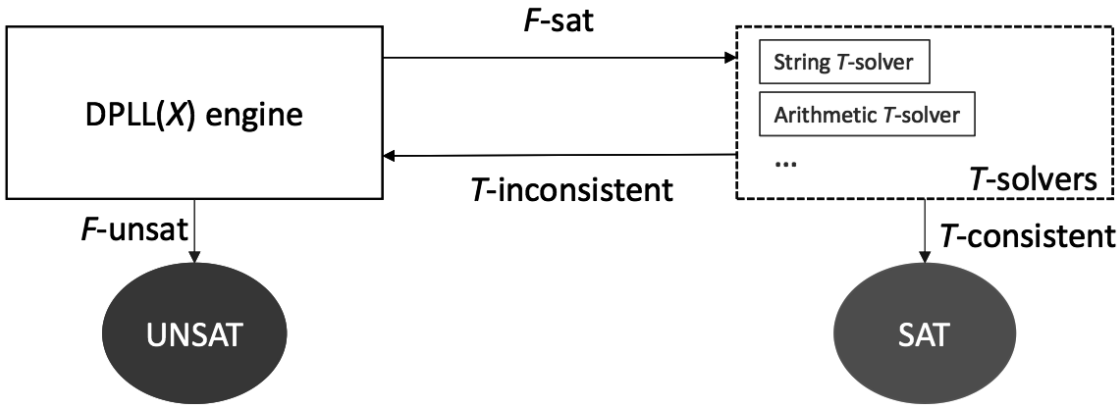

Figure 3: A general DPLL(T) Architecture.

4.3 The DPLL(T) Architecture

The DPLL(T) architecture can be divided into two parts, as shown in Figure 3 : the logic solving part and the theory solving part. In the logic solving part, it usually refers to the DPLL-based SAT solver with the capability to handle incremental clause assertions.

The theory solving part usually contains several dedicated solvers for background theories, called T-solvers. Compared to a generic theory solver, a T-solver requires the mechanisms for incrementality and backtracking. A T-solver maintains a set of T-related literals. This set is a subset of the partial assignment M.

Initially, the SAT solver gets the formula F from the input. It tries to find a model for the literals. If failed, it returns *unsat*; otherwise, it distributes each literal in M to a corresponding T-solver according to some T-signature-based heuristics.

When a T-solver gets a set of literals, it first checks whether these literals are consistent with the theory. If it is T-consistent, the T-solver does nothing but reports consistent back to the main engine. If it is M-inconsistent, it either returns a conflict (in a form of literal conjunction), or propagates a new literal with an explanation (in a form of implication).

If all T-solvers report T-consistent, the procedure stops and answers *sat* with the model M; otherwise, the SAT solver collects all clauses returned by T-solvers, and asserts them into the formula F. Then, it tries to build a model from this new formula.

4.4 First order theory

In this paper, we are working in the context of many-sorted first-order logic with equality. A *theory* is a pair $\mathcal{T} = (\Sigma\ \mathtt{I})$ where Σ is a signature and $\mathtt{I}$ is a class of Σ-interpretations, that is closed under variable reassignment. A Σ-formula φ is $satisfiable$ (resp., $unsatisfiable$) in $\mathcal{T}$ if it is satisfied by some interpretation in $\mathtt{I}$. A set Γ of formulas entails in $\mathcal{T}$ a Σ-formula φ, written $\Gamma \models_T \varphi$, if every interpretation in $\mathtt{I}$ that satisfies all formulas in Γ satisfies φ as well. The set Γ is $satisfiable$ in T if $\Gamma \models_T \bot$ where $\bot$ is the universally false atom. We will write $\Gamma \models \varphi$ to denote that Γ entails φ in the class of all Σ-interpretations. We will use $\approx$ as the (infix) logical symbol for equality, which has type $\sigma \times \sigma$ for all sorts σ in Σ and is always interpreted as the identity relation. We write $s \not\approx t$ as an abbreviation of $\neg s \approx t$.

5 A Theory of Strings and Regular Language Membership

We use **Str**, **Lan** and **Int** to denote the String sort, the Language sort and the Integer sort, respectively. We use Σ_{SRL_p} to denote the signature with three sorts **Str**, **Lan** and **Int**. $\mathcal{T}_{SRL_p}$ refers to the theory of strings with length and positive regular language membership constraints over a signature Σ_{SRL_p}. The interpretations of $\mathcal{T}_{SRL_p}$ differ only on the variables. They all interpret **Int** as the set of integer numbers $\mathbb{Z}$, **Str** as the set of all strings $\mathbb{S}$ over some fixed finite alphabet $\mathcal{A}$ of characters, and **Lan** as the power set of $\mathcal{A}^*$. The signature includes the following predicate and function symbols: The common symbols (e.g., $+, -, \leq$) of linear integer arithmetic are interpreted as usual. The signature of the sort **Str** consists the following symbols:

- a constant symbol, or string constant, for each word of $\mathcal{A}^*$, interpreted as word;
- a function symbol con: $String \times \cdots \times String \to String$, interpreted as the word concatenation;
- a function symbol len: $String \to Int$, interpreted as the word length function;

The signature of the sort **Lan** consists the following symbols:

- a function symbol set: $String \to Lan$, interpreted as the function mapping each word $w \in \mathcal{A}^*$ to the language $\{w\}$;
- a function symbol star: $Lan \to Lan$, interpreted as the Kleene clousre operator;
- a predicate symbol in: $String \to Lan$, interpreted as the membership predicate;
- a suitable set of additional function symbols: $union, inter, rempty, allchars, optrange, loop, plus, comp$, interpreted as language concatenation, conjunction and so on;

We define three types of terms:

- *string term* : any term of sort **Str** or of the form $(len\ x)$;
- *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. We define three types of constraints:

- *string constraint* : is a (dis)equality $(\neg)s \approx t$ with s and t are string terms;
- *arithmetic constraint* : is a (dis)equality $(\neg)s \approx t$ or $(\neg)s > t$ where s and t are arithmetic terms;
- *RL constraint* : is a literal of the form $(s \mathrm{in} r)$ where s is a string term and r is a regular expression.

Note that if x and y are string variables, len x is both a string and an arithmetic term and $(\neg)$len $x \approx$ len y is both a string and an arithmetic constraint. A $\mathcal{T}_{SRL_p}$-constraint is a string, arithmetic or RL constraint. We will use $\models_{SRL_p}$ to denote entailment in $\mathcal{T}_{SRL_p}$.

6 A Calculus for $\mathcal{T}_{SRL_p}$

Here we describe the essence of the combined solver for $\mathcal{T}_{SRL_p}$ abstractly and declaratively, as a tableaux-style calculus. The solver can be understood as a specific proof procedure for the calculus. The current solver treats RL constraints in a fairly naive way. In particular, the Kleene star operator is processed by unrolling:

$$(s \text{ in star } r) \xrightarrow{\text{is reduced to}} s = \epsilon \vee s \approx \text{con}(x, y) \wedge (x \text{ in } r) \wedge (y \text{ in star } r)$$

where x and y are fresh variables. This approach makes the solver non-terminating in general over such constraints. In our description below we focus only on the derivation rules that deal with string and arithmetic constraints.

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

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

Two terms s, t are equivalent iff $s \approx t \in \mathcal{C}(\mathrm{S})$. For all $t \in \mathcal{T}(\mathrm{S})$, we denote its equivalence class in E_S by $[t]_S$. Here E_S is an equivalence relation over $\mathcal{T}(\mathrm{S})$ induced by the constraints in S.

$$\begin{aligned}\texttt{con}(\mathbf{s}, \texttt{con}(\mathbf{t}), \mathbf{u}) &\to \texttt{con}(\mathbf{s}, \mathbf{t}, \mathbf{u}) \\ \texttt{con}(\mathbf{s}, \epsilon, \mathbf{u}) &\to \texttt{con}(\mathbf{s}, \mathbf{u}) \\ \texttt{con}(s) &\to s \\ \texttt{con}() &\to \epsilon\end{aligned}$$

$$\begin{aligned}\texttt{con}(\mathbf{s}, c_1 \cdots c_i, c_{i+1} \cdots c_n, \mathbf{u}) &\to \texttt{con}(\mathbf{s}, c_1 \cdots c_n, \mathbf{u}) \\ \texttt{len}(\texttt{con}(s_1, \cdots, s_n)) &\to \texttt{len}(s_1) + \cdots + \texttt{len}(s_n) \\ \texttt{len}(c_1, \cdots, c_n) &\to n\end{aligned}$$

Figure 4: Normalization rewrite rules for terms

Configurations. The calculus operates over *configurations* consisting of the distinguished configuration unsat and of tuples of the form ⟨S, A, R, F, N, C, B⟩ where

- S, A, R are respectively a set of string, arithmetic, and RL constraints;
- F is a set of pairs $s \mapsto \mathbf{a}$ where $s \in \mathcal{T}(\mathrm{S})$ 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 E_S and $\mathbf{a}$ is a tuple of atomic string terms;
- C is a set if terms of sort **Str**;
- B is a set of *buckets* where each bucket is a set of equivalence classes of E_S.

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

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

In an *initial configuration* all constraints are distributed according to their signature among the components S, A, R and the other components are empty. For convenience, we assume that the S component of the initial configuration contains an equation $x \approx t$ for each non-variable term $t \in \mathcal{T}(\mathrm{S})$, where x is a variable of the same sort as t. We also assume that all terms in the initial configuration are reduced with respect to the rewrite rules.We denote by $t\downarrow$ the normal form of a term t with respect to the rewrite rules. For example, $(x, \epsilon, c_1, c_2c_3, y)\downarrow = (x, c_1c_2c_3, y)$.

Before we describe the derivation rules, we define few properties of a configuration.

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

Invariant 1. The proof procedures that we will describe maintain these invariants on the derivable configurations of the form ⟨S, A, R, F, N, C, B⟩:

1 All terms are reduced with respect to the rewrite system in Figure4.

2 S is a partial map from $\mathcal{T}(\mathtt{S})$ to normalized tuples of atomic terms.

3 S is a partial map from E_S to normalized tuples of atomic terms.

4 For all terms s where $[s] \to (a_1, \cdots, a_n) \in N$ or $s \to (a\text{–}1, \cdots, a_n) \in F$, we have $S \models_{SLR_p} s \approx \mathtt{con}(a_1, \cdots, a_n)$ and $S \models a_i \not\approx \epsilon for i = 1, \cdots, n$.

5 For all $B_1, B_2 \in \mathtt{B}, [s] \in B_1$ and $[t] \in B_2, S \models \mathtt{len}\ s \approx \mathtt{len}\ t$ iff $B_1 = B_2$.

6 C contains only reduced terms of the form $\mathtt{con}(\mathbf{a})$.

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

We say that a derivable configuration ⟨S, A, R, F, N, C, B⟩ is *saturated*, if

- N is a total map over E_S.
- B is a partition of E_S, and
- it is saturated with respect to all rules other than Reset.

Derivation Rules. The rules of the calculus are provided in Figures 5 through 9 in *guarded assignment form*. A derivation rule applies to a configuration K if all of the rule's premises hold for K. A rule's conclusion describes how each component of K is changed, if at all. We write S, t as an abbreviation for $S \cup \{t\}$. Rules with two conclusions, separated by the symbol ‖, are non-deterministic branching rules. We treat a string constant l in a tuple of terms indifferently as term.For example, a tuple $(x, c_1c_2c_3, y)$ with the three-character constant $c_1c_2c_3$ will be seenal so as the tuple (x, c_1, c_2c_3, y),(x, c_1c_2, c_3, y) or (x, c_1, c_2, c_3, y). All equalities and disequalities in the rules are treated modulo symmetry of ≈. We assume the availability of a procedure for checking entailment in the theory of linear integer arithmetic ($\models_{LIA}$) and one for computing congruence closures and checking entailment in EUF ($\models$).

The first four rules in Figure 5 describe the interaction between arithmetic reasoning and string reasoning, achieved via the propagation of entailed constraints in the shared language. R-Star is the only rule for handling RL constraints that we provide here. We chose it because the constraints matching its premise can be generated, by rule F-Loop in Figure 8, even if the initial configuration contains no RL constraints. The basic rules for string constraints are shown in Figure 6. The functionality and rationale of the last three should be straightforward. Reset is meant to be applied after the set S changes since in that case normal and flat forms may need updating. S-Cycle identifies a concatenation of terms with one them when the remaining ones are all equivalent to ϵ.

The bulk of the work is done by the rules in Figures 7 and 8. Those in Figure 7 compute an equivalent flat form (consisting of a sequence of atomic terms) for all non variable terms that are not in the set C. Flat forms are used in turn to compute normal forms as follows. When all terms of an equivalence class e except for variables and terms in C have the same flat form, that form is chosen by N-Form1 as the normal form of e. When an equivalence class e consists only of variables and terms in C, one of them is chosen by N-Form2 as the normal form of e. The first two rules of Figure 8 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.

The rules in Figure 9 are used to put equivalence classes of terms of sort **Str** into buckets based on the expected length of the value they will be given eventually by a satisfying assignment. The main idea is that different equivalence classes go into different buckets (using D-Base) unless they have the same length. In the latter case, they go into the same bucket only if we can tell they cannot have the same value (using D-Add). D-Split is used to reduce the problem to one of the two previous cases. The goal is that, on saturation, each bucket B can be assigned a unique length n_B, and each equivalence class in B can evaluate to a unique string constant of that length. Card makes sure that n_B is big enough to have enough string constants of length n_B.

Proof Procedure. A proof procedure is defined by the repeated application of the calculus rules according to the six steps below (also as shown in Figure 10). When applying a branching rule the procedure tries the left-branch configuration first. It interrupts a step and restarts with Step 0 as soon as a constraint is added to S. The procedure keeps cycling through the steps until it derives a saturated configuration or the unsat one. In the latter case, it continues with another configuration in the derivation tree, if any.

Step 0 *Reset* : Apply Reset to reset buckets, and flat and normal forms.That is ⟨B, F, N⟩ = ⟨ϕ, ϕ, ϕ⟩.

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

$$\mathbf{A-Prop}\frac{S \models \text{len } x \approx \text{len } y}{\text{A} := \text{A}, \text{len } x \approx \text{len } y}$$

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

$$\mathbf{Len}\frac{x \approx t \in \mathcal{C}(S)\ x \in \mathcal{V}(S)}{\text{A} := \text{A}, \text{len } x \approx (\text{len } t)\downarrow}$$

$$\mathbf{Len-Split}\frac{x \in \mathcal{V}(S \cup A)\ x : Str}{\text{S} := \text{S}, x \approx \epsilon \parallel \text{A} := \text{A}, \text{len } x > 0}$$

$$\mathbf{A-Conflict}\frac{A \models_{LIA} \bot}{unsat}$$

$$\mathbf{R-Star}\frac{s\ in\ star(set\ t) \in R\ s \not\approx \epsilon \in \mathcal{C}(S)}{\text{S} := \text{S}, s \approx \text{con}(t, z)\text{R} := \text{R}, z\ in\ star\ (set\ t)}$$

Figure 5: Rules for theory combination, arithmetic and RL constraints. The letter z denotes a fresh Skolem variable.

$$\mathbf{S-Cycle}\frac{t = \text{con}(t_1, \cdots, t_i, \cdots, t_n)\ t \in \mathcal{T}(S) \setminus C\ t_k \approx \epsilon \in \mathcal{C}(S) \text{ for all } k \in \{1, \cdots, n\} \setminus \{i\}}{\text{S} := \text{S}, t \approx t_i\ C := C(C, t) \setminus \{t_i\}}$$

$$\mathbf{Reset}\frac{}{\text{F} := \phi, \text{N} := \phi, \text{B} := \phi}$$

$$\mathbf{S-Split}\frac{x, y \in \mathcal{V}(S)\quad x \approx y, x \not\approx y \in \mathcal{C}(S)}{\text{S} := \text{S}, x \approx y \parallel S := S, x \not\approx y}$$

$$\mathbf{S-Conflict}\frac{x \approx t \in \mathcal{C}(S)\quad s \not\approx t \in \mathcal{C}(S)}{\text{unsat}}$$

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

Figure 6: Basic string derivation rules

$$\mathbf{F-Form1}\frac{t = \mathrm{con}(t_1, \cdots, t_n)\ t \in \mathcal{T}(S) \setminus (\mathcal{D}(F) \cup C)\ \ N[t_1] = s_1 \cdots N[t_n] = s_n}{\mathrm{F} := \mathrm{F}, t \mapsto (s_1, \cdots, s_n)\downarrow}$$

$$\mathbf{F-Form2}\frac{l \in \mathcal{T}(S) \setminus \mathcal{D}(F)}{\mathrm{F} := \mathrm{F}, t \mapsto (l)}$$

$$\mathbf{N-Form1}\frac{[x] \notin \mathcal{D}(N)\ \ s \in [x] \setminus (C \cup \mathcal{V}(S))\ \ Ft = Fs \text{for all } t \in [x] \setminus (C \cup \mathcal{V}(S))}{\mathrm{N} := \mathrm{N}, [x] \mapsto Fs}$$

$$\mathbf{N-Form2}\frac{[x] \notin \mathcal{D}(N)\ \ [x] \subseteq C \cup \mathcal{V}(S)}{\mathrm{N} := \mathrm{N}, [x] \mapsto (x)}$$

Figure 7: Normalization derivation rules. The letter l denotes a string constant.

$$\mathbf{F-Unify}\frac{F\ s = (w, u, u_1)\ \ F\ t = (w, u, v_1) s \approx t \in \mathcal{C}(S)\ S \models \mathrm{len}\ u \approx \mathrm{len}\ v}{\mathrm{S} := \mathrm{S}, u \approx v}$$

$$\mathbf{F-Split}\frac{\begin{array}{c}F\ s = (w, u, u_1)\ \ F\ t = (w, u, v_1) s \approx t \in \mathcal{C}(S)\ S \models \mathrm{len} u \approx \mathrm{len}\ v \\ u \notin \mathcal{V}(v_1)\ v \notin \mathcal{V}(u_1)\end{array}}{\mathrm{S} := \mathrm{S}, u \approx con(v, z) \parallel S := S, v \approx con(u, z)}$$

$$\mathbf{F-Loop}\frac{\begin{array}{c}F\ s = (w, x, u_1)\ \ F\ t = (w, u, v_1, x, v_2)\ s \approx t \in \mathcal{C}(S)\ x \notin \mathcal{V}((v, v1)) \\ \mathrm{S} := \mathrm{S}, x \approx con(z_2, z),\ con(v, v1) \approx con(z_2, z_1),\ con(u_1) \approx con(z_1, z_2, v_2) \\ \mathrm{R} := \mathrm{R}, z\ in\ star(set\ con(z_1, z_2))\ \ \mathrm{C} := \mathrm{C}, t\end{array}}{}$$

Figure 8: Equality reduction rules. The letters z, z_1, z_2 denote fresh Skolem variables.

$$\mathbf{D-Base}\frac{s \in \mathcal{T}(S)\ \ s : \mathrm{Str}\ \ S \models \mathrm{len}\ s \approx \mathrm{len}_B \text{ for no } B \in \mathrm{B}}{\mathrm{B} := \mathrm{B}, \{[s]\}}$$

$$\mathbf{Card}\frac{B \in \mathrm{B}\ \ |B| > 1}{\mathrm{A} := \mathrm{A}, \mathrm{len}_B > \lfloor log_{|\mathcal{A}|}\ (|B| - 1) \rfloor}$$

$$\mathbf{D-Add}\frac{\begin{array}{c}s \in \mathcal{T}(S)\ \ s : \mathrm{Str}\ \ \mathrm{B} = \mathrm{B'}, B\ S \models \mathrm{len}\ s \approx \mathrm{len}_B[s] \notin B \\ \text{for all } e \in \mathrm{B} \text{ there are } w, u, u_1, v, v_1 \text{ such that} \\ (N[s] = (w, u, u_1), Ne = (w, v, v1),\ \ S \models \mathrm{len}\ u \approx \mathrm{len} v, u \not\approx v \in \mathcal{C}(S))\end{array}}{\mathrm{B} := \mathrm{B}', (B \cup \{[s]\})}$$

$$\mathbf{D-Split}\frac{\begin{array}{c}s \in \mathcal{T}(S)\ \ s : \mathrm{Str}\ \ \mathrm{B} = \mathrm{B'}, B\ S \models \mathrm{len}\ s \approx \mathrm{len}_B[s] \notin B\ e \in B \\ (N[s] = (w, u, u_1), Ne = (w, v, v1),\ \ S \models \mathrm{len}\ u \not\approx \mathrm{len}\ v\end{array}}{\mathrm{S} := \mathrm{S}, u \approx con(z_1, z_2), len\ z_1 \approx len\ v \parallel \mathrm{S} := \mathrm{S}, v \approx con(z_1, z_2), len\ z_1 \approx len\ u}$$

Figure 9: Disequality reduction rules. Letters z_1, z_2 denote fresh Skolem variables. For each bucket $B \in \mathrm{B}, len_B$ denotes a unique term len x where $[x] \in B$. $|-|$ denotes the cardinality operator.

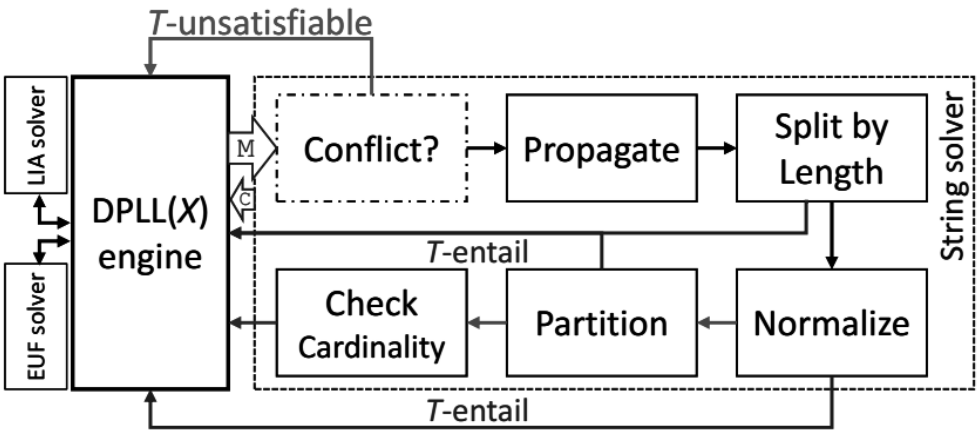

Figure 10: Abstracted core proof procedure for string.

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

Step 3 *Compute Normal Forms for Equivalence Classes* : Apply `S-Cycle` to completion and then the rules in Figure 7 to completion. If this does not produce a total map N, there must be some $s \approx t \in \mathcal{C}(S)$ such that $F\ s$ and $F\ t$ have respectively the form (w, u, u_1) and (w, v, 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 $\texttt{len}x \approx \texttt{len}y$ nor $\texttt{len}x \not\approx \texttt{len}y$, apply `L-Split` to them; otherwise, apply any applicable rules from Figure 8, giving preference to `F-Unify`.

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

Step 5 *Add length constraint for cardinality* : Apply the rule `Card` for all buckets B in `B`, which adds an arithmetic constraint corresponding to the minimal length of terms in B based on the number of equivalence classes in B and the cardinality $|\mathcal{A}|$ of the alphabet.

Here we try to illustrate the procedure's workings with a couple of examples.

Example 1: Suppose we have the input constraints: $A = \phi$ and $S = \{\texttt{len}(x) \approx \texttt{len}(y), x \not\approx \epsilon, z \not\approx \epsilon, \texttt{con}(x, l_1, z) \approx \texttt{con}(y, l_2, z)\}$, where l_1, l_2 are distinct constants of the same length.

After a failure of applying `S-Conflict` and `A-Conflict` for a possible conflict, the proof procedure applies `Len` and `Len-Split` to completion. All resulting derivation tree branches except one can be closed with `S-Conflict`. In the leaf of the non-closed branch every string variable is in a disequality with ϵ. In that configuration, the string equivalence classes are $\{x\}, \{y\}, \{z\}, \{l_1\}, \{l_2\}, \{\epsilon\}$, and $\{\texttt{con}(x, l_1, z), \texttt{con}(y, l_2, z)\}$. 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 $\texttt{F con}(x, l_1, z) = (x, l_1, z)$ and $\texttt{F con}(y, l_2, z) = (y, l_2, z)$. Then `F-Unify` is applied to add the equality $x \approx y$ to S. After this, the procedure restarts but now with the string equivalence classes $\{x, y\}, \{z\}, \{l_1\}, \{l_2\}, \{\epsilon\}$, and $\{\texttt{con}(x, l_1, z), \texttt{con}(y, l_2, z)\}$.After similar steps as before, the terms in the last equivalence class get the flat forms (x, l_1, z) and (x, l_2, z) respectively (assuming, without loss of generality, x is chosen as the representative term for $\{x, y\}$). Using `F-Unify` again, the procedure adds the equality $l1 \approx l2$ to S and then derives `unsat` with `S-Conflict`. This closes the derivation tree, showing that the input constraints are unsatisfiable. □

Example 2: Suppose now the input constraints are: $A = \phi$ and $S = \{\texttt{len}(x) \approx \texttt{len}(y), x \not\approx \epsilon, z \not\approx \epsilon, \texttt{con}(x, l_1, z) \not\approx \texttt{con}(y, l_2, z)\}$, with l_1, l_2 are distinct constants of the same length.

After similar steps as in previous example, the procedure can derive a configuration where the string equivalence classes are $\{x\}, \{y\}, \{z\}, \{l_1\}, \{l_2\}, \{\epsilon\}, \{\texttt{con}(x, l_1, z)\}$ and,$\{\texttt{con}(y, l_2, z)\}$. After computing normal forms for these classes, it attempts to construct a partition `B` of them into buckets. However, notice that if it adds [x],say, to `B` using `D-Base`, then neither `D-Base` (since $S \models \texttt{len}\ x \approx \texttt{len}\ y$) nor `D-Add` (since $x \not\approx y \notin \mathcal{C}(S)$) is applicable

to $[y]$. So it applies S-Split to x and y. In the branch where $x \approx y$, the procedure subsequently restarts, and computes normal forms as usual. At that point it succeeds in making B a partition of the string equivalence classes, by placing $[\texttt{con}(x, l_1, z)]$ and $[\texttt{con}(y, l_2, z)]$ into the same bucket using D-Add, which applies because their corresponding normal forms are (x, l_1, z) and (x, l_2, z) respectively. Any further rule applications lead to branches with a saturated configuration, each of which indicates that the input constraints are satisfiable. □

Example 3: Suppose now the input constraints are: $A = \phi$ and $S = \{\texttt{con}(x, a) \approx \texttt{con}(b, x)\}$, where a, b are distinct constants of length 1.

After applying the rule F-Loop, two fresh variables y and z are introduced,and the constraints $a \approx \texttt{con}(y, z)$ and $b \approx \texttt{con}(z, y)$ are asserted to S. These two constraints already generate a conflict in S. □

7 Correctness

Here we briefly formalize the main correctness properties of the calculus. Since the solver can be seen as a specific proof procedure, it immediately inherits those properties (*refutation soundness* and *solution soundness*). This means in particular that when the solver terminates with a sat or unsat output, that output is corrects.

Theorem 7.1 (Refutation Soundness) *For any closed derivation tree rooted by an initial configuration $\langle S_0, A_0, R_0, \phi, \phi, \phi, \phi\rangle$, the set $S_0 \cup A_0 \cup R_0$ is unsatisfiable in $\mathcal{T}_{SRL_p}$.*

Theorem 7.2 (Solution Soundness) *If a derivation tree with initial configuration $\langle S_0, A_0, R_0, \phi, \phi, \phi, \phi\rangle$ contains a saturated configuration then the set $S_0 \cup A_0$ is satisfiable in $\mathcal{T}_{SRL_p}$. The saturated configuration induces a satisfying assignment for the set $S_0 \cup A_0$ in $\mathcal{T}_{SRL_p}$.*

Since the decidability problem in $\mathcal{T}_{SRL_p}$ is still an open question. The authors did not provide any proof that the calculus is *refutation complete*. However there is different strategy or approach which is solution complete. That is if a set of constraints is satisfiable, the calculus is able to find a solution. However, the calculus(a different version) is solution complete but not necessarily refutation complete, it is not guaranteed to terminate if a set of constraints is unsatisfiable. More detail on the correctness properties and the proofs of these theorem can be found in **??**.

8 Implementation in DPLL(T)

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

As we have described the calculus, the literals of an assignment M are partitioned into string constraints (corresponding to the set S), arithmetic constraints(the set A) and membership constraints (the set R). These sets are subsequently given to three independent solvers, in our case they are: the string solver, the arithmetic solver, and the membership solver. The rules A-Prop and S-Prop model the standard mechanism for Nelson-Oppen theory combination, where entailed equalities are communicated between these solvers. The satisfiability check performed by the arithmetic solver is modeled by the rule A-Conflict. Since there is no additional requirement on the arithmetic solver, so a standard DPLL(T) theory solver for linear integer arithmetic can be used. The behavior of the membership solver is described by the rule R-Star. The remaining rules model the behavior of the string solver.

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

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

In DPLL(T), when a theory solver determines that M is unsatisfiable (in the solver's theory) it generates a *conflict clause*, the negation of an unsatisfiable subset of M. The string solver maintains a compact representation of $\mathcal{C}(S)$ at all times.

To construct conflict clauses, the string solver also maintains an explanation $\psi_{s,t}$ for each equality $s \approx t$, when the equality is added to S by applying S-Cycle, F-Unify or standard congruence closure rules. The explanation $\psi_{s,t}$ is

a conjunction of string constraints in M such that $\psi_{s,t} \models_{SRL_p} s \approx t$. For `F-Unify`, the string solver maintains an explanation ψ for the flat form of each term $t \in \mathcal{D}(F)$ where $\psi \models_{SRL_p} t \approx \texttt{con}(\texttt{F}\ t)$. When a configuration is determined to be unsatisfiable by `S-Conflict`,that is, when $s \approx t, s \not\approx t \in \mathcal{C}(\texttt{S})$ for some s, t, it replaces the occurrence of $s \approx t$ with its corresponding explanation ψ, and then replaces the equalities in ψ with their corresponding explanation, and so on, until ψ contains only equalities from M. Then it reports as a conflict clause (the clause form of) $\psi \Rightarrow s \approx t$.

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

9 Evaluation

According to the authors, a theory solver based on the calculus and proof procedure described in the previous section within is implemented as part of SMT solver CVC4. The string alphabet $\mathcal{A}$ for this implementation is the set of all 256 ASCII characters. To evaluate they solver an experimental comparison with two of the string solvers was performed. These other solvers are Z3-STR and Kaluza. These solvers, which have been widely used in security analysis.

For our evaluation 47,284 benchmark problems from a set of about 50K benchmarks generated by Kudzu were used. Kudzu is a symbolic execution framework for Javascript, and available on the Kaluza website. Then these benchmarks were translated into CVC4's extension of the SMT-LIB 2 format and into the Z3-STR format. The CVC4, Z3-STR and Kaluza was run on each benchmark with a 20-second CPU time limit.

The results provided strong evidence that CVC4's string solver was sound. They also provided evidence that `unsat` answers from Z3-STR and Kaluza for problems on which CVC4 times out couldnot be trusted. They also showed that CVC4's string solver answered sat more often than both Z3-STR and Kaluza, providing a correct solution in each case. Thus, it is claimed to the best tool for both satisfiable and unsatisfiable problems.

10 Conclusion

In this paper, we tried to presented the overview of a automatic reasoning tool for solving quantifier-free constraints over unbounded strings with length and regular language membership. This approach allows to integrates a specialized theory solver for such constraints within the DPLL(T) framework. The authors have given experimental evidence that the implementation of their idea in the SMT solver CVC4 is highly competitive with existing tools. However, it would be better to support a richer language of string constraints that occur often in practice, especially in security applications. Such expressiveness would promote the use of automatic reasoning tool to increase the quality of modern software system.

References

[1] Tianyi Liang, Andrew Reynolds, Cesare Tinelli, Clark Barrett, and Morgan Deters. A DPLL(T) theory solver for a theory of strings and regular expressions. In Armin Biere and Roderick Bloem, editors, *Proceedings of the 26th International Conference on Computer Aided Verification*, volume 8559 of *Lecture Notes in Computer Science*. Springer, 2014.

[2] Cesare Tinelli and Clark Barrett. *CVC4 is an efficient open-source automatic theorem prover.* `http://cvc4.cs.nyu.edu/web`.

[3] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. *SMT-LIB Satisfiability Modulo theories library.* `http://smtlib.cs.uiowa.edu/`.

[4] *CVC4 wiki.* `http://cvc4.cs.nyu.edu/wiki/Strings`.

[5] Nikolaj Bjørner, Nikolai Tillmann, and Andrei Voronkov. Path feasibility analysis for string-manipulating programs.

[6] G S Makanin. The problem of solvability of equations in a free semigroup. *Mathematics of the USSR-Sbornik*, 32(2):129, 1977.

[7] Minh-Thai Trinh, Duc-Hiep Chu, and Joxan Jaffar. S3: A symbolic string solver for vulnerability detection in web applications.

[8] Tianyi Liang. Automated reasoning over string constraints. *PhD Dissertation, Department of Computer Science, The University of Iowa, Dec 2014.*