SMT for Strings

Seminar: Satisfiability Checking

SS 2015

Meshkatul Anwer
Supervision: Cornelius Aschermann

July 17, 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 present an introduction over the DPLL($\mathcal{T}$) procedure and architecture. The reader can skip this section, if the concepts are clear to them. In the subsequent sections 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($\mathcal{T}$) 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 problem (SAT) is to answer whether there is an assignment for those variables, such that the formula is evaluated to be `true`. 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 `true` 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. This is known as 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.

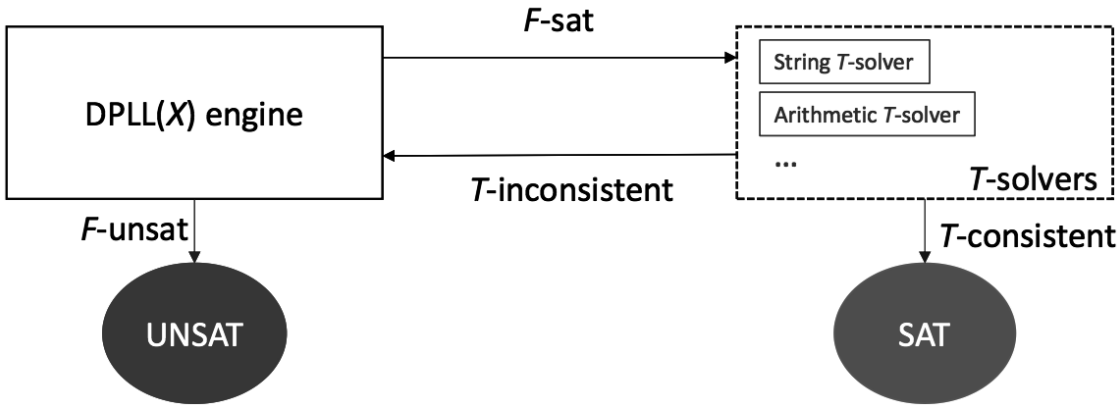

Figure 3: A general DPLL(T) Architecture. The image is taken from [8].

4.2 The DPLL($\mathcal{T}$) Architecture

In Boolean formulas, the signature only contains propositional variables and logical connectives.But 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 $\mathcal{T}$ as a $\mathcal{T}$-solver. Note that a $\mathcal{T}$-solver can only handle conjunctions of literals. Given a formula over several theories where each theory has a $\mathcal{T}$-solver, the Nelson-Oppen combination [9] provides a procedure to reason about this constraint.

The DPLL($\mathcal{T}$) 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 $\mathcal{T}$-solvers. Compared to a generic theory solver, a $\mathcal{T}$-solver requires the mechanisms for incrementality and backtracking. A $\mathcal{T}$-solver maintains a set of $\mathcal{T}$-related literals. This set is a subset of the partial assignment $\mathcal{M}$.

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

When a $\mathcal{T}$-solver gets a set of literals, it first checks whether these literals are consistent with the theory. If it is $\mathcal{T}$-consistent, the $\mathcal{T}$-solver does nothing but reports consistent back to the main engine. If it is $\mathcal{M}$-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 $\mathcal{T}$-solvers report $\mathcal{T}$-consistent, the procedure stops and answers `sat` with the model $\mathcal{M}$; otherwise, the SAT solver collects all clauses returned by $\mathcal{T}$-solvers, and asserts them into the formula $\mathcal{F}$. Then, it tries to build a model from this new formula.

5 The Theory solver $\mathcal{T}_{SL}$

5.1 Objective

We want a solver to handle a finite set of (unbounded)string constraints, length constraints and regular language memberships. Most of the other solvers available in the market try to solve the constraints by converting them into constraints of other available theories. Here we will see, the algebraic approach taken by the authors of cvc4 [2] to implement the solver. The solver is described as a rule-based procedure by the original authors in [1]. Before we go into detail, we will present few formal preliminaries and definitions. We will focus our discussion only on solving string constraints and length constraints.

5.2 Basic

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 $\neg s \approx t$. These notations are defined by the original authors in [1] and we will also use them with the same context in the subsequent sections.

5.3 Core language

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 $(s \text{ in } r)$ where s is a string term and r is a regular expression.

5.4 The foundation

We have a finite set of constraints of type string constraints and arithmetic constraints. The string solver takes this set as input and checks for satisfiability. If it finds a satisfying solution, it reports the set of constraints is satisfiable otherwise reports as `unsat`. From now on we will only consider string constraints and arithmetic constraints. 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 $\mathcal{T}_{SL}$-constraint is a string, arithmetic constraint. We will use $\models_{SL}$ to denote entailment in $\mathcal{T}_{SL}$. Now we will present few definitions, which are defined by author in [1]. These definitions are used for the detail description of the decision procedure.

Definition 5.1 *congruence closure of S*

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

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

This equation is taken from [1].

Definition 5.2 *equivalence relation*

Iff $s \approx t \in \mathcal{C}(S)$, where $s,t \in \mathcal{T}(S)$, the terms s,t are called *equivalent*. For all $t \in \mathcal{T}$(S), its equivalence class in E_S is denoted by $[t]_S$. Where E_S is an *equivalence relation* over $\mathcal{T}(S)$ induced by the constraints in S.

Definition 5.3 *normalized form*

A term is in *normalized form*, if it can not be reduced any more respect to the rewrite rules. The rewrite rules are shown in Figure 4. For example, $(x, \epsilon, c_1, c_2c_3, y)$ becomes $(x, c_1c_2c_3, y)$ after the reduction.

$$\texttt{con}(\mathbf{s}, \texttt{con}(\mathbf{t}), \mathbf{u}) \rightarrow \texttt{con}(\mathbf{s}, \mathbf{t}, \mathbf{u})$$
$$\texttt{con}(\mathbf{s}, \epsilon, \mathbf{u}) \rightarrow \texttt{con}(\mathbf{s}, \mathbf{u}) \qquad \texttt{con}(\mathbf{s}, c_1 \cdots c_i, c_{i+1} \cdots c_n, \mathbf{u}) \rightarrow \texttt{con}(\mathbf{s}, c_1 \cdots c_n, \mathbf{u})$$
$$\texttt{con}(s) \rightarrow s \qquad \texttt{len}(\texttt{con}(s_1, \cdots, s_n)) \rightarrow \texttt{len}(s_1) + \cdots + \texttt{len}(s_n)$$
$$\texttt{con}() \rightarrow \epsilon \qquad \texttt{len}(c_1, \cdots, c_n) \rightarrow n$$

Figure 4: Normalization rewrite rules for terms. The rules are taken from [1]

Definition 5.4 *configurations*

A *configuration* is defined in [1] as a tuple 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 $s \in \mathcal{T}$(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. For each bucket $B \in$ B, len_B denotes a unique term `len` x, where $[x] \in B$.

Initially the sets S, A, R store the input problem and grow with the introduction of new constraints derived by derivation rules. 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. The procedure 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.

A configuration is called *saturated* by author in [1], if

- `N` is a total map over E_S.
- `B` is a partition of E_S, and

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

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

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

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

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

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

Figure 5: Rules for the combination of string, arithmetic and RL constraints. The letter z denotes a fresh Skolem variable.The rules are taken from [1] and they are in their original form.

- It is not possible to apply any derivation rule on the terms other than `Reset`.

The decision procedure concludes `sat` when it reaches such a configuration in the derivation tree.

Definition 5.5 *derivation rule*

A *derivationrule* applies to a *configuration* K, if all of the rule's premises hold. A rule's conclusion describes how each component of the *configuration* K is changed, if at all. An application of rule may produce two or more beaches in the derivation tree. These rules have the symbol $\parallel$ in their conclusion. The author in [1] called them as are non-deterministic branching rules. However, in the implementation the branching is applied from left to write. According to the author the derivation rules are only applicable on any configuration when it maintain the following conditions as *Invariant*:

- All terms are reduced with respect to the rewrite system in Figure 4.

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

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

- 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 $S \models_{SLR_p} \ s \approx \texttt{con}(a_1, \cdots, a_n)$ and $S \models a_i \not\approx \epsilon for i = 1, \cdots, n$.

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

- `C` contains only reduced terms of the form $\texttt{con}(\mathbf{a})$.

In the paper [1] the authors have presented a list of the derivations rules. These rules are listed in Figures from 5 to 9. Due to space restriction, we will only describe few of them bellow.

The first four rules `A-Prop`, `S-Prop`, `Len` and `Len-Splite` in Figure 5 describe the interaction between arithmetic solver and string solver. This is achieved via the propagation of entailed constraints on the shared terms. The rule `A-Conflict` derives `unsat` if the arithmetic part of the constraints unsatisfiable. The basic rules for string constraints e.g. `S-Cycle`, `S-Split`, `S-Conflict` are shown in Figure 6. The functionality of `S-Split`, `L-Split` is straightforward. `S-Conflict` derives `unsat` if $\mathcal{C}(S)$ contains both equality and dis-equality between the same pair of strings. `S- Split` tries to guess whether two strings are equal or not, while `L-Split` tries to guess whether two string variables have the same length. Application of split rules causes the branching of the derivation tree. `Reset` is meant to be applied when new constraints are introduced to the set S. The major part of the work is done by the *normalization derivation rules* and *equality reduction rules*. These rules shown in Figures 7 and 8. Detail about the individual rule can be found in [8] and in [1].

The *disequality reduction rules* in Figure 9 are used to partition the equivalence classes of terms of sort **Str** into buckets. This classification is done on the expected length of the value. The main target is that, on saturation, each bucket B can be assigned a unique length n, and each equivalence class in B can evaluate to a unique string constant of that length. These assignments willl be used as model. Finally, the rule `Card` makes sure that n is big enough to have enough string constants of length n. The brief description of the rules presented here is taken from [8].

Definition 5.6 *derivation tree*

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

$$\textbf{Reset}\frac{}{\mathrm{F}:=\phi,\mathrm{N}:=\phi,\mathrm{B}:=\phi}$$

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

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

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

Figure 6: Basic string derivation rules. The rules are taken from [1] and they are in their original form.

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

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

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

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

Figure 7: Normalization derivation rules. The letter l denotes a string constant.The rules are taken from [1] and they are in their original form.

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

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

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

Figure 8: Equality reduction rules. The letters z, z_1, z_2 denote fresh Skolem variables.The rules are taken from [1] and they are in their original form.

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

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

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

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

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.The rules are taken from [1] and they are in their original form.

A *derivation tree* 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. 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 this section we have presented all the definitions we need to finally describe the procedure. All these definitions are taken from the paper [1].

5.5 Proof Procedure

.

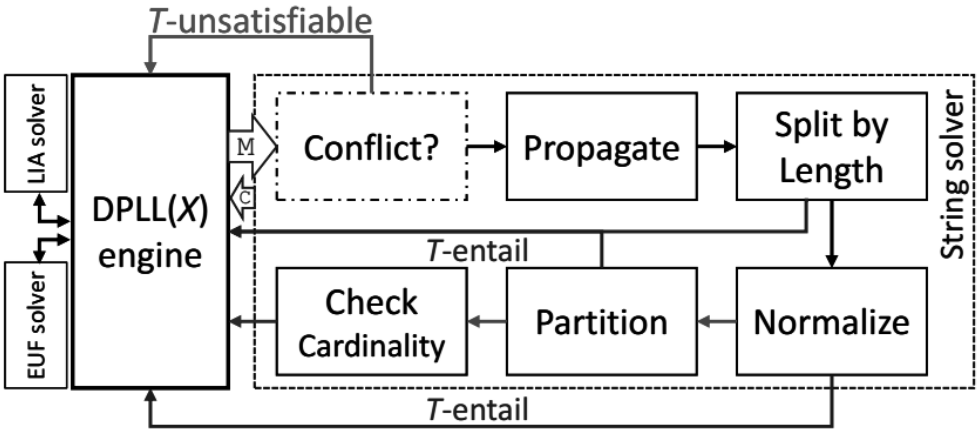

Figure 10: Abstracted core proof procedure for string. The figure is taken from [8].

In this section, we will see how the derivation rules are applied to produce the derivation tree. According to the authors of paper [1] this proof procedure is a highly abstracted version of the one which is implemented in cvc4 string solver. The main procedure is based on the repeated application of the rules according to the seven steps below (also as shown in Figure 10). In cases of split, the left branch configuration is tried first. The procedure interrupts and restarts with Step 0 as soon as new constraint is introduced to S. The procedure keeps cycling through the steps until it derives a configuration where no rules applies or the `unsat` one. In the case of `unsat`, if there is other branch configuration, the procedure continues with that branch in the derivation tree.

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

Step 1 : *Check for conflicts* : Apply `S-Conflict` or `A-Conflict` if the configuration is unsatisfiable due to the current string or arithmetic constraints.

Step 2 : *Propagate* : Apply `S-Prop` and `A-Prop` and propagate entailed equalities between S and A.

Step 3 : *Add length constraints* : For each non-variable $t \in \mathcal{T}(S)$, apply `Len` to add the equality $\texttt{len}(x) \approx \texttt{len}(t)$ to `A`. For each variable x in $\mathcal{V}(S \cup A)$, apply `Len-Split`,and then first explore the branch where $x \approx \epsilon$.

Step 4 : *Compute Normal Forms for Equivalence Classes* : Apply `S-Cycle` to shrinks concatenation terms. Then apply the normalization rules `F-Form1`,`F-Form2`, `N-Form1` and `N-Form2` to completion. The idea is to produce a total map N. If not, then rules like `L-Split`, `F-Unify`, `F-Split`, `F-loop` are applied according to different cases. Detail on this can be found on paper [8].

Step 5 : *Partition equivalence classes into buckets* : The target is to distribute each equivalence class to a bucket. First apply `D-Base` and `D-Add` to completion. This should make B a partition of E_S. If not, then there is an equivalence class $[x]$ which is not contained in any bucket. At this step rules like `S-Split`, `D-Split`, `L-Split` are applied according to different cases. Detail on this can be found on paper [8].

Step 6 : *Add length constraint for cardinality* : Apply the rule `Card` for each bucket. This will introduce new arithmetic constraints 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. For instance, if Σ is a finite alphabet of 256 characters and S entails that 257 distinct strings of length 1 exist, then S is unsatisfiable.

In this section, we have presented a brief and abstract overview of the procedure. The description of the different steps are taken from [8]. In the subsequent section we will present few examples of the application of the derivation according to this procedure.

5.6 Examples

Now we try to illustrate the procedure's workings with respect of two example. First example shows how the procedure conclude *unsat* for a set of constraints which are unsatisfiable. And the second shows how the procedure finds a saturated configuration for a set of constraints which are satisfiable. Both the examples are taken from [1].

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

The procedure starts with *step* 0 by applying `Reset`. It tries to find conflicts by applying `S-Conflict` and `A-Conflict`. Since the arithmetic constraint set A is empty and there are no contradicting string constraints in S, *check for contradiction* fails.

The procedure tries the step *propagate*. For this example this step does not introduce any new constraint in S, as A is empty. Now the procedure applies `Len` and `Len-Split` repeatedly as long as there is possible application. The application of these two splitting rules causes the branching of the derivation tree. However, all branches are closed by rule `S-Conflict` expect one. This is shows in Figure as the left branches of each configuration.

In this non closing configuration, the string equivalence classes are $\{x\}, \{y\}, \{z\}, \{l_1\}, \{l_2\}, \{\epsilon\}$, and $\{\texttt{con}(x, l_1, z), \texttt{con}(y, l_2, z)\}$. Now the procedure goes on to compute normal forms by applying `N-Form2`, `F-Form2` and `N-Form1`.The flat forms `F con`$(x, l_1, z) = (x, l_1, z)$ and `F con`$(y, l_2, z) = (y, l_2, z)$ are computed by `F-Form1`. Then `F-Unify` is applied to add the equality $x \approx y$ to S. This causes the procedure to restart, but with an extended constraints set. Which induces a new 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, the procedure reached a stage where it computes the flat forms (x, l_1, z) and (x, l_2, z) for the corresponding equivalence class, by choosing x as representative of y. At this point the procedure applies rule `F-Unify` again. This introduces a new equality $l_1 \approx l_2$ to S and eventually derives `unsat` with `S-Conflict`. That is the procedure ended up with a derivation tree where each branch is closed and conclude that the input constraints are unsatisfiable. The resulting derivation tree is shown in figure .

Example 2: 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.

Similar to the previous example, the procedure reaches a configuration where the string equivalence classes are $\{x\}, \{y\}, \{z\}, \{l_1\}, \{l_2\}, \{\epsilon\}, \{\texttt{con}(x, l_1, z)\}$ and,$\{\texttt{con}(y, l_2, z)\}$. In this case, the procedure attempts to partition the the equivalence classes into buckets. It fails to create the full partition, as rule `D-Base` nor rule `D-Add` is applicable to $[y]$ because of $S \models \texttt{len}\ x \approx \texttt{len}\ y$ and $x \not\approx y \notin \mathcal{C}(S)$.

In such a case the procedure tries to guess by applying rule `S-Split` to x and y. It case two branches in the derivation tree, one for $x \approx y$ and another for $x \not\approx y$.

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 as new constraints are introduced to S. Eventually the procedure computes normal forms and succeeds in making a full partition of `B`. The procedure places $[\texttt{con}(x, l_1, z)]$ 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. Thus the procedure concludes that the problem is satisfiable. For each of the branches with saturated configuration satisfying models can be generated.

Here in this section we have tried to explore the inner working of the decision procedure. According to the authors [1], the presented version the procedure is a simplified one. The procedure and derivation rules, which are implemented as part of cvc4 is much more complex and elaborate.

5.7 Integration into SMT solver architecture

In this section we will explain, how the string theory solver is integrated into modern DPLL(T) framework. We will briefly discuss the issues of *theory propagation*, *lemma learning* and *conflict analysis*.

Modern SMT solvers combine a SAT solver with multiple specialized *theory solvers*. The SAT solver maintain an evolving set F of clauses and a set M of literals representing partial assignment. The whole process is derived by the SAT solver. It periodically consults the theory solver, to find whether M is satisfiable in its theory. The literals of an assignment M are partitioned into string constraints, arithmetic constraints. These sets are subsequently given to the independent solvers. The rules `A-Prop` and `S-Prop` model the mechanism for *theory combination*. This mechanism is known as

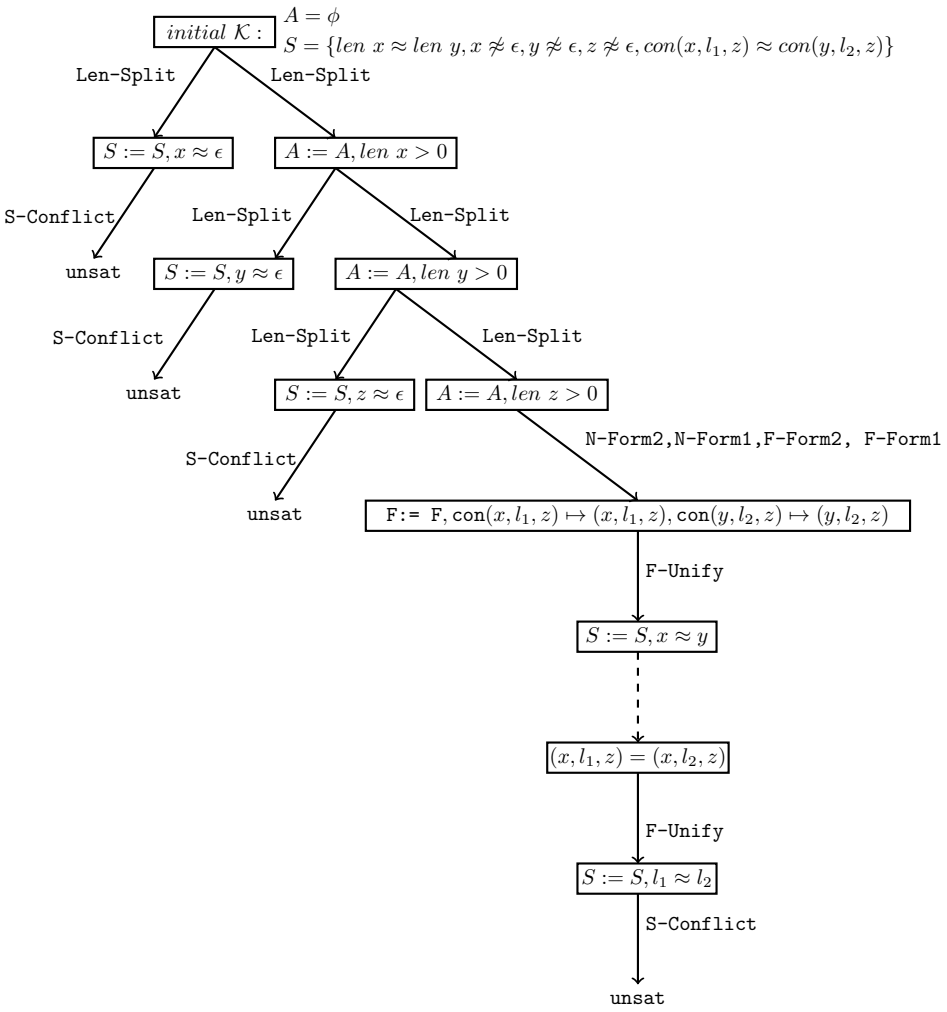

Figure 11: The derivation tree for example 1. Here l_1, l_2 are distinct constants of same length.

Nelson-Oppen theory combination [9], where the entailed equalities are communicated between multiple solvers. The rule `A-Conflict` modeled the satisfiability checking done by the arithmetic solver. As there is no additional requirement for the arithmetic solver, a standard theory solver for linear integer arithmetic can be used.

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. This approach of *lemma learning* is very efficient as new lemmas are introduced only when it is needed. Similarly, the rules `Len`, `Len-Split`, and `Card` are involved in introduction of new constraints into `A`. This is done by the string solver by adding lemmas to F containing arithmetic constraints. For example, 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.

When a theory solver determines that M is unsatisfiable, it produces a *conflict clause* to support the decision. A *conflict clause* is the negation of an unsatisfiable subset of M. Whenever the string solver introduces any equality to `S`, it also maintains an explanation $\psi_{s,t}$ for each equality $s \approx t$. An explanation $\psi_{s,t}$ is a conjunction of string constraints in M such that $\psi_{s,t} \models s \approx t$. When a configuration is decided to be unsatisfiable by rule `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. Finally, it reports the conflict clause $\psi \Rightarrow s \approx t$.

The core idea of the above explanations are taken from [1].

5.8 Correctness

Here we briefly discus the 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*). The procedure is *refutation sound*, that is when the procedure answers `unsat`, it can be trusted even for strings of unbounded length. And the procedure is *solution sound*, that is when the procedure answers *sat*, it can be trusted. In the original paper [1], the authors have claimed that they have a version of the procedure which is also *solution complete*, that is when the procedure answers *sat*, it will eventually get a model by finite model finding. The authors did not provide any proof that the procedure is *refutation complete*, that is when the procedure answers *unsat*, it is not guaranteed to drive refutation. More detail on the correctness properties and the proofs of these theorem can be found in [8].

6 Evaluation

According to the authors [1], a theory solver based on the calculus and proof procedure described in the previous sections, is implemented as part of SMT solver CVC4. The string alphabet $\mathcal{A}$ for this implementation is the set of all 256 ASCII characters. An experimental comparison with two other the string solvers was conducted. The other string solvers are Z3-STR [10] and Kaluza[11]. These two solvers are widely used in security analysis. For the evaluation, 47,284 benchmark problems from a set of about 50K benchmarks generated by Kudzu [12] were used. These benchmarks were translated into CVC4's extension of the SMT-LIB 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. In the paper [1], it is claimed 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. More on the evaluation can be found in [8].

7 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($\mathcal{T}$) 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 such automatic reasoning tool in commercial software constructions. Eventually, help us to build secure and high quality correct software.

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.*

[9] Greg Nelson and Derek C. Oppen. Simplification by cooperating decision procedures. *ACM Trans. Program. Lang. Syst.*, 1(2):245–257, October 1979.

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

[11] Steve Hanna Stephen McCamant Prateek Saxena, Devdatta Akhawe and Dawn Song. *Kaluza, the string solver for the core of Kudzu.* `http://webblaze.cs.berkeley.edu/2010/kaluza/`.

[12] Prateek Saxena, Devdatta Akhawe, Steve Hanna, Feng Mao, Stephen McCamant, and Dawn Song. A symbolic execution framework for javascript.