SMT for Strings

Seminar: Satisfiability Checking

SS 2015

Meshkatul Anwer
Supervision: Cornelius Aschermann

July 21, 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. In this paper we will describe a solver for the theory of unbounded strings with concatenation and length.

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 analysis.

The ability to reason about string values is a major task in the field of security analysis. 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 a SMT solver for strings. The ideas are presented in paper [1] and a practical version of the solver is implemented into the SMT solver CVC4 [2] core.

2 Security analysis with theory solvers

In the context of security analysis, we want to find out whether security flaws exist in a piece of program. To answer such questions, we express security specification and properties as a set constraints in some background theories. Then this set of constraints is provided to an automatic reasoning tool, which should answer whether the constraints are satisfiable or not. The usual intension is to get the bug revealing inputs reported by the tool. The conventional manual testing or verification processes may not be compete. As they usually model partial scenarios. Whereas the automatic reasoning tool facilitates the search of whole

space for possible flaws. Thats why the approach of verification with the help SMT solvers is quite reliable, efficient and complete.

Example: Here is an example of such constraint,

$$assert(x = \texttt{con}(\text{"}ab\text{"}, z) \wedge (y = \texttt{con}(\text{"}de\text{"}, z) \vee y = \texttt{con}(\text{"}abc\text{"}, l)) \wedge x = y \wedge \texttt{len}(x) > 6) \quad (1)$$

where `con` and `len` are functions with the usual meaning of string concatenation function and string length function and x, y, z and l are variables of type string. Figure 1 shows how this formula can be encoded into the language of theory solver.

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

(declare-fun x () String)
(declare-fun y () String)
(declare-fun z () String)
(declare-fun l () String)

(assert (= x (str.++ "ab" z)))
(assert (or (= y (str.++ "de" z)) (= y (str.++ "abc" l))) )
(assert (= x y))
(assert (> (str.len x) 6))

(check-sat)
(get-value (x y z l) )
```

Figure 1: The formula 1 is encoded into smt-lib[3] format.

When this constraint is provided to the theory solver and asked for satisfiability check, it answers with a satisfying solution like,

$$x = \text{"}abcAAAA\text{"},\ y = \text{"}abcAAAA\text{"},\ z = \text{"}cAAAA\text{"} \text{and } l = \text{"}AAAA\text{"}$$

The example shows how complex constraints can be expressed. Especially in the context of security analysis we want to express complex constrains over strings and other data types. The ability to reason over strings and other data types is very important. Many security analysis tools exists in the industry and used for practice. However, they are either able to reason on certain theories or they are very limited in expressiveness. In this paper, we will present a general purpose theory solver for strings and length constraints.

3 Related work

There has been a lot of research effort on the development of the efficient theory solver for Strings [4]. Theoretically the general world equation problem or the satisfiability problem of theory of strings(with most of common string functions) is undecidable [5]. However, a restricted theories of strings is decidable and practically useful. Different approaches to solve the problem have been proposed and implemented. A class of solvers is based on reducing the string constraints to constraints in other theories e.g. theory of bit-vector. Examples of such solver are Hampi [6] and Kaluza [7]. However, they can only solve problem over fixed length strings.

In this paper, we will present a SMT solver for strings based an algebraic approach. The whole decision procedure is described as a set of derivation rules and their application strategy. The procedure allows to express the constraints over unbounded strings. A practical theory solver over string based on this approach has been implemented into the CVC4 SMT solver[2] core. The detail of the description can be found in the paper [1]. We will try to explain the main ideas presented by the authors.

4 Background

In this section, we will present an introduction over the SMT problem solving. Later we will focus on the theory solver part.

4.1 The SAT problem

In classical propositional logic, a boolean formula is constructed from a set of variables connected by logical operators like $\wedge, \vee, \neg$. The boolean variables can be assigned a truth value that is either `true` or `false`. The classical *SAT problem*is defined as, given a boolean formula, is there any assignment, such that the formula is evaluated to `true`. There are many decision procedures to solve this classic SAT problem. Among them, the DPLL [8] procedure is most frequently used in most modern SAT solvers.

4.2 The SMT Problem

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`. For example,

$$s_1 \wedge (s_2 \vee s_3) \wedge s_4 \wedge s_5$$

is a boolean formula for the example 1. Where each s_i is just a representation for each constraint. The DPLL procedure does not need to interpreted the actual meaning of the constraint. When the set $\{s_1, s_2 \vee s_3, s_4, s_5\}$ is given as input, the DPLL procedure returns partial assignment.

$$\{s_1,\ (s_2 \vee s_3),\ s_4,\ s_5\} \mapsto \{s_1,\ s_2,\ s_4,\ s_5\} \text{ or } \{s_1,\ s_3,\ s_4,\ s_5\}$$

Now this partial assignments are need to be checked for satisfiability. Each boolean variable in the partial assignments represents a constraint. The constraints may have function symbols (e.g. `con`, `len`) or predicate symbols. The meaning of the variables, constants, predicates and functions must be defined. 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 only be interpreted in some background theories. We say an SMT formula is satisfiable if there exists a model that satisfies both the logical formula and the background theories. In our case the relevant background theories are $\mathcal{T}_{SL}$ and $\mathcal{T}_{LIA}$. We are focusing on the solver for the theory $\mathcal{T}_{SL}$. We will use $\mathcal{T}_{SL}$-solver to denote the theory solver for strings.

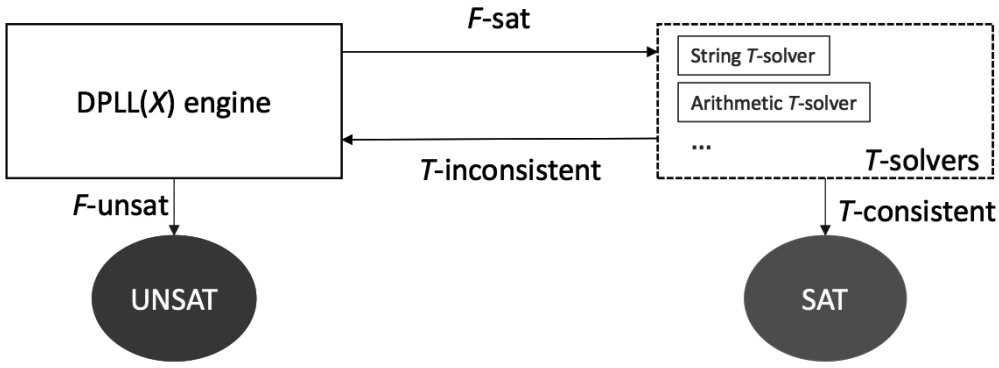


Figure 2: An overview of the interaction between SAT core and $\mathcal{T}$-solvers. The image is sourced from [9].

The SMT problem solving procedure can be divided into two parts, as shown in Figure 2 : 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.

Initially, the SAT solver gets the formula as input. It tries to find a model for the literals. If fails, it returns `unsat`; otherwise, it distributes each literal to a corresponding $\mathcal{T}$-solver according to the signature of the literal.

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 SAT core. Otherwise, it either returns a conflict, or propagates a new literal. For example, in case of our example, when the partial

model $\{s_1,\ s_2,\ s_4,\ s_5\}$ is given to the $\mathcal{T}_{SL}$-solver. It finds a contraction $s_1 \wedge s_2 \rightarrow \neg s_4$ and reports back to the core.

If all $\mathcal{T}$-solvers report $\mathcal{T}$-consistent, the procedure stops and answers `sat` with the current partial assignments as model ; otherwise, the SAT solver collects all clauses returned by $\mathcal{T}$-solvers, and asserts them into the formula. Then, it tries to build a model from this new formula. For example, in case of our example, the core add $\neg s_2$ as a new fact and starts over again.

5 The Theory Solver

In this section we will focus on the SMT solver for strings, which can handle a finite set of (unbounded)string constraints and length constraints.

5.1 Core Language

We use Σ_{SL} to denote the signature over the String sort and the Integer sort. The common symbols (e.g., $+, -, \leq$) of linear integer arithmetic are interpreted with their usual meaning. The signature defines a finite set of alphabets $\mathcal{A}$ (e.g. 256 ascii characters). A valid string will have characters only from $\mathcal{A}$. The terms of sort string can be :

- constants symbol e.g. $'a', 'ab', 'abc', 'helloworld'$. That is any word from $\mathcal{A}^*$.
- variables symbol e.g. x, y, z.
- function symbol con: $String \times \cdots \times String \rightarrow String$, interpreted as the word concatenation;
- a function symbol len: $String \rightarrow Int$, interpreted as the word length function;

We have two types of constraints depending on the signature:

- *string constraint* : is a (dis)equality $(\neg)s \approx t$ with s and t are string terms; e.g. $x = \texttt{con}(\text{"}ab\text{"}, z)$, $y = \texttt{con}(\text{"}de\text{"}, z)$, $y = \texttt{con}(\text{"}abc\text{"}, l)$, $x = y$.
- *length constraint* : is a (dis)equality $(\neg)s \approx t$ or $(\neg)s > t$ where s and t are arithmetic terms; e.g. `len` $x > 6$.

Throughout the section we will use S to denote the set of constraints over strings terms and A to denote set of arithmetic constraints. For example,

$$S = \{x = \texttt{con}(\text{"}ab\text{"}, z), y = \texttt{con}(\text{"}abc\text{"}, l), x = y.\}, A = \{\texttt{len}\ x > 6.\}$$

where, x, y, z, l are variables of type string. 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 $\mathcal{T}_{SL}$.

5.2 The decision procedure

The decision procedure is defined as a set of derivation rules. The procedure starts with an initial configuration keeps applying the derivation rules according to the strategy. During this repeated application of rules a derivation tree is produced. When any rule is selected for application, it may cause the tree to branch at this configuration. These rules are usually called spiting rules. A configuration may be `unsat`. If the procedure reaches a configuration which is `unsat`, it will backtrack and choose another branch for rule application. If there is no other branch in case of `unsat`, we call the branch is closed. Another possibility is that, the procedure reaches a configuration, where no more rule application is possible. Such a configuration is a *staturated* one. A derivation tree is called *closed* if all of its branches are closed. If the procedure ends up with a closed derivation tree, it concludes that the set of constraints are unsatisfiable or declares `unsat`. On the other hand, If the procedure ends up with a saturated configuration, it concludes that the set of constraints are satisfiable or declares `unsat`. Whenever any new string constraint is derived from any rule application, the procedure restart.

At the start of the procedure, each term in the set of constraints must be in their normalized form according to the reduction rules listed below.

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

Here is few examples of terms and their normal forms, $\texttt{con}(\text{"}ab\text{"}, \texttt{con}(\text{"}c\text{"}, l)) \rightarrow \texttt{con}(\text{"}abc\text{"}, l)$, $\texttt{len}(\text{"}abc\text{"}) \rightarrow 3$, $\texttt{len}(\texttt{con}(\text{"}abc\text{"}, \text{"}def\text{"})) \rightarrow 6$.

The procedure starts applying rules on the initial configuration according to the following steps:

1. *Check conflicts* : The idea is to drive conflict as early as possible. Apply `S-Conflict` or `A-Conflict` to check if the configuration is unsatisfiable due to the current string or arithmetic constraints.

$$\texttt{A-Conflict}\frac{A \models_{LIA} \bot}{\text{unsat}}$$
$$\texttt{S-Conflict}\frac{s \approx t \in S \quad s \not\approx t \in S}{\text{unsat}}$$

   Examples of the application are: $S : \{\cdots, x \approx \epsilon, x \not\approx \epsilon, \cdots\} \rightarrow \texttt{unsat}$, $S : \{\cdots, x \approx y, x \not\approx y, \cdots\} \rightarrow \texttt{unsat}$, $A : \{\texttt{len}(x) \approx \texttt{len}(y), \texttt{len}(x) \not\approx \texttt{len}(y)\} \rightarrow \texttt{unsat}$

2. *Propagate* : Introduce new constraints induced by constraints in other theories (e.g. $\mathcal{T}_{LIA}$ and $\mathcal{T}_{SL}$).

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

   Examples of the application are: $S \models (\texttt{len}\ x \approx \texttt{len}\ y) \rightarrow A := A, \texttt{len}\ x \approx \texttt{len}\ y$, $S \models (\texttt{len}\ x \approx \texttt{len}\ \text{"}abc\text{"}) \rightarrow A := A, \texttt{len}\ x = 3$

3. *Split by Length* : Introduce new constraint to the arithmetic constraints for equalities in string constraints. Introduce branching for free variables in string constraints with `Len-Split`.

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

   Examples of the application are: $S \models (x \approx y) \rightarrow A := A, \texttt{len}\ x \approx \texttt{len}\ y$, $S \models (x \approx \text{"}abc\text{"}) \rightarrow A := A, \texttt{len}\ x = 3$

4. *Normalize* : Compute the normalized form for each term. As in the previous steps we may have added new constraints to the set of constraints we started with. So these terms should be normalized as per the normalization reduction rules. Apply `S-Cycle` to shrink the concatenation terms. Then apply the normalization rules `F-Form1`, `F-Form2`, `N-Form1` and `F-Form2` to the completion. The idea is to produce a total map of normal forms. If still there exists any term, which is not in its normalized form first apply splitting and then finally unify.

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

Examples of the application is : $\texttt{con}(x, m) \approx \texttt{con}(y, n), \texttt{len}(x) \approx \texttt{len}(y) \rightarrow S := S, x \approx y$.

5. *Partition* : Each equivalence class should be in their corresponding group (bucket). The target is to distribute each equivalence class to a bucket. This classification is done on the expected length of the value. First apply `D-Base` and `D-Add` to completion. If the partition is not compete, splitting is required for the free variables. That is , at this point of rule application, there exits an equivalence class which is not contained in any bucket. At this step rules like `S-Split`, `D-Split`, `L-Split` are applied according to different cases.

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

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

Otherwise the procedure have reached a saturated configuration.

6. *Check cardinality* : This is the last step of the procedure. The procedure has reached a saturated configuration. Now for each bucket B the procedure 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.

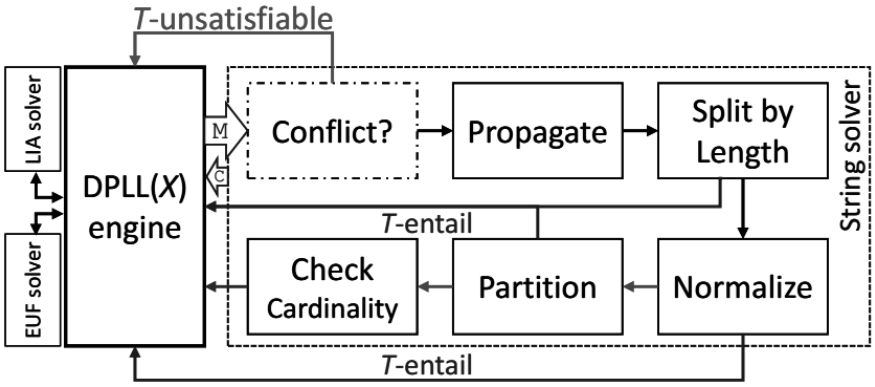


Figure 3: Abstracted core proof procedure for string. The figure is sourced from [9].

Figure 3 shows the overall application strategy. The full list of derivation rules can be found in Figures from 4 to 8. The four rules `A-Prop`, `S-Prop`, `Len` and `Len-Splite` in Figure 4 describe the interaction between arithmetic solver and string solver. This is achieved via the propagation of entailed constraints on the shared terms. This technique of cooperation between different theory solvers is well know as Nelson-Oppen method [10]. The rule `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. Whenever any new constraints are introduced to the set S, the procedure restart. Finally, when the procedure reached the saturated configuration, the rule `Card` makes sure that each bucket has been assigned big enough length. Detail about the individual rule can be found in [9] and in [1].

6 Example

In this section, we will use the example formula 1 to explain how the decision procedure works. Consider the five constraints s_1, s_2, s_3, s_4, s_5 on the left side of Figure 9. Since the SAT core cannot interpret the

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

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

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

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

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

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

Figure 4: The rules for theory combination[1]. `R-Start` is handle cases with regular expression.

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

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

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

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

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

Figure 5: Basic string derivation rules [1].

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

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

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

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

Figure 6: The rules for normalization[1].

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

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

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

Figure 7: The rules for equality reduction [1]. The rule `F-Loop` is used to detect looping problem.

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

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

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

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

Figure 8: The rules for dis-equality reduction[1].

$$s_1 : x = \texttt{con}(\text{"}ab\text{"}, z)$$
$$s_2 : y = \texttt{con}(\text{"}de\text{"}, z)$$
$$s_3 : y = \texttt{con}(\text{"}abc\text{"}, l)$$
$$s_4 : x = y$$
$$s_5 : \texttt{len}\ x > 6$$
$$assert(s_1 \wedge (s_2 \vee s_3) \wedge s_4 \wedge s_5)$$

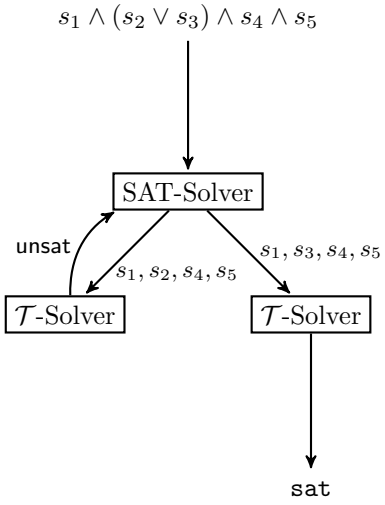


Figure 9: On the left the set of constraints and the boolean formula. And on the right side the interaction with SAT-Solver core and theory solver is shown.

constraints, it treats them as five independent boolean variables and tries to assign values to them. The core starts by setting s_1, s_4 and s_5 to `true`. Then the SAT core choose true for s_2 before s_3. Now the selected constraints $\{s_1, s_2, s_4, s_5\}$ are forwarded to the theory solver. This is shown as the left branch of the tree on the right side of Figure 9. The theory solver will answer `unsat` for this set of constraints. The SAT core backtracks and tries the other option. That is the SAT core choose s_3 and ask theory solver to solve the new set of constraints $\{s_1, s_3, s_4, s_5\}$. This is shown as the right branch of the tree on the right of Figure 9. The theory solver eventually finds a satisfying solution in this branch.

Left derivation tree: The input constraints are: $A = \{\texttt{len}\ x > 6\}$ and $S = \{x = \texttt{con}('ab', z), y = \texttt{con}('de', z), x = y\}$.

The procedure starts with by applying `Reset`. Then it tries to find conflict by applying `S-Conflict` and `A-Conflict` and fails to find any contradiction in the current constraints.

The procedure tries to apply `A-Prop` and `S-Prop`. The constraints in different theories fail to induce new constraint for other.

Now the procedure keeps trying to apply rules `Len` and `Len-Split`. The application of these split rule causes the tree to branch into new configurations. However, all the branches are closed with respect to the rule `S-Conflict`. This is shown in the left branches in Figure 10.

Now for the non closing configuration, the procedure goes on to compute the normal forms. $x = y$ causes $\texttt{con}('ab', z), \texttt{con}('de', z)$ to be classified as equivalent term. Further application of rule `F-Unify` causes to introduce $'ab' =' de'$ into S and eventually this branch get closed by `S-Conflict`. Thus the procedure decides that, the given set of constraints are unsatisfiable.

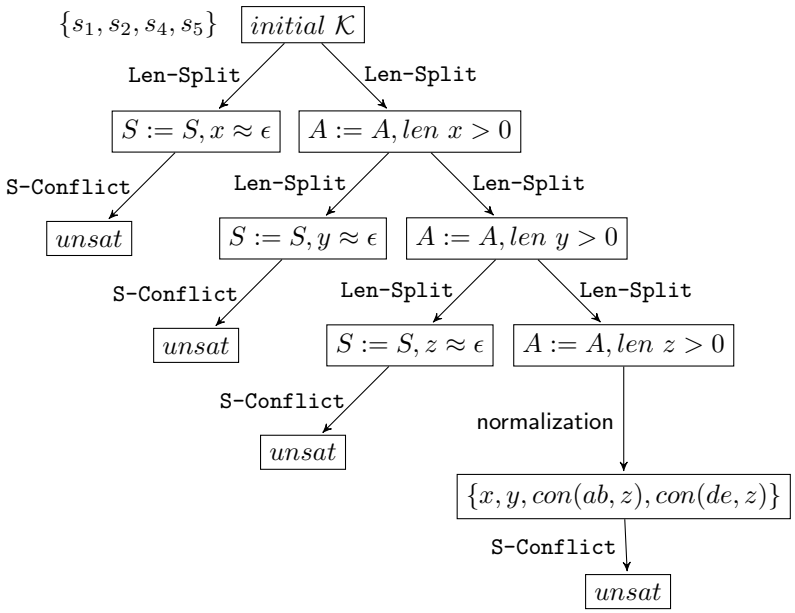
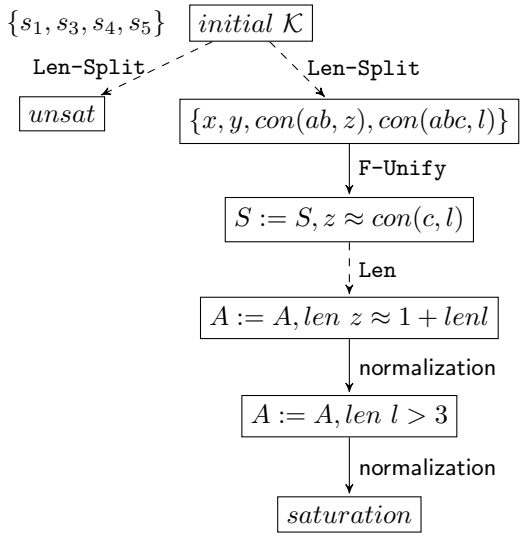


Figure 10: The branch of derivation tree, which ends up in `unsat`.



$$l \approx "AAAA" \mapsto z \approx "cAAAA" \mapsto x, y \approx "abcAAAA"$$

Figure 11: The branch of derivation tree, which ends up in saturation.

Right derivation tree: The input constraints are: $A = \{\texttt{len}\ x > 6\}$ and $S = \{x = \texttt{con}('ab', z), y = \texttt{con}('abc', l), x = y\}$. The derivation tree is shown in Figure 11.

Similar to the left derivation tree, the procedure reaches a configuration, where the string equivalence classes are like $\{x, y, \texttt{con}('ab', z), \texttt{con}('abc', l)\}, \{z\}, \{l\}$. Application of the rule `F-Unify` causes the introduction of new constraints $z = \texttt{con}('c', l)$. The procedure restarts with new equivalence classes $\{x, y, \texttt{con}('ab', z), \texttt{con}('abc', l)\}, \{z, \texttt{con}('c', l)\}, \{l\}$ with the application of `Reset`. The new constraint in S induces new constraints in A. That application of `Len` causes $A := A, \texttt{len}\ z = 1 + \texttt{len}\ l$. At this point the procedure starts to compute the normal forms by applying different rules. Eventually the constraint $\texttt{len}\ l > 3$ is computed and a saturated configuration is reached. As no more rule application is not possible at this configuration. Thus the procedure concludes that the problem is satisfiable.

Finally, for each buckets an arithmetic constraint on the minimum length is added. For example, $\texttt{len}\ l > 3$ causes the length of the bucket which contain l to be 4. Now it is trivial to find a satisfying assignment. that if $l =' AAAA'$ then $z =' cAAAA'$ and $x, y =' abcAAAA'$.

7 Correctness

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. However, the procedure is *refutation complete*, that is for an unsatisfiable set of constraints the procedure may not terminate. More detail on the correctness properties and the proofs of these theorem can be found in [9].

8 Evaluation

According to the authors [1], a theory solver based 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 [11] and Kaluza [7]. 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. In the paper [1], it is claimed that, CVC4's string solver performed better. More detail on the evaluation can be found in [9].

9 Conclusion

In this paper, we tried to present the overview of a automatic reasoning tool for solving constraints over unbounded strings with length. We have found, the idea of implementing the string solver natively is unique. This approach allowed high interaction with core of CVC4 SMT solver. So the performance is better than others. It also allows any general purpose arithmetic solver be easily integrated. However, the application of the rules is complicated and very hard to implement. 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.

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

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

[6] Adam Kiezun, Vijay Ganesh, Philip J. Guo, Pieter Hooimeijer, and Michael D. Ernst. Hampi: A solver for string constraints. In *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis*, ISSTA '09, pages 105–116, New York, NY, USA, 2009. ACM.

[7] 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/`.

[8] Martin Davis and Hilary Putnam. A computing procedure for quantification theory. *J. ACM*, 7(3):201–215, July 1960.

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

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

[11] 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.

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