SMT for Strings

Seminar: Satisfiability Checking

Meshkatul Anwer
Supervision: Cornelius Aschermann

SS 2015

RWTH AACHEN UNIVERSITY

Introduction

- Web service security is important.
- String is the main information carrier.
- String reasoning is important.
- Here we will present : SMT for strings.

Objectives

- We want a string solver:
  - Fast
  - Robust
  - Expressive(Regexp)
  - Other Theories (Int, Bool)

Constraints:examples

- The language for $\mathcal{T}_{SL}$:
  - over a finite set of alphabets $\mathcal{A}$ (e.g. 256 ASCII characters)
  - terms can be constants e.g. `"a"`, `"ab"`, `"abc"`,`"helloWorld"`, ...
  - terms can be free constants or variables (e.g. `x`, `y`, `z`,...)
  - terms can be String concatenation:
    `con` : $String \times \cdots \times String \rightarrow String$
  - terms can be length terms: `len` : $String \rightarrow Int$

Constraints:examples

- examples of String constraints:
  - $s_1 : \mathtt{x} = \mathtt{con}(\mathtt{"ab"}, \mathtt{z})$
  - $s_2 : \mathtt{y} = \mathtt{con}(\mathtt{"de"}, \mathtt{z})$
  - $s_3 : \mathtt{y} = \mathtt{con}(\mathtt{"abc"}, \mathtt{l})$
  - $s_4 : \mathtt{x} = \mathtt{y}$

Constraints:examples

- examples of String constraints:
  - $s_1 : \mathtt{x} = \mathtt{con}(\mathtt{"ab"}, \mathtt{z})$
  - $s_2 : \mathtt{y} = \mathtt{con}(\mathtt{"de"}, \mathtt{z})$
  - $s_3 : \mathtt{y} = \mathtt{con}(\mathtt{"abc"}, \mathtt{l})$
  - $s_4 : \mathtt{x} = \mathtt{y}$
- example of length constraints:
  - $s_5 : \mathtt{len}(\mathtt{x}) > 6$
- example of the Boolean formula:
  - $assert(s_1 \wedge (s_2 \vee s_3) \wedge s_4 \wedge s_5)$

Constraints:encoding into smtlib

```
(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: the formula encoded in smt-lib 2 format

constraints:

- s_1 : $\mathtt{x} = \mathtt{con}(\mathtt{"ab"}, \mathtt{z})$
- s_2 : $\mathtt{y} = \mathtt{con}(\mathtt{"de"}, \mathtt{z})$
- s_3 : $\mathtt{y} = \mathtt{con}(\mathtt{"abc"}, \mathtt{l})$
- s_4 : $\mathtt{x} = \mathtt{y}$
- s_5 : $\mathtt{len}(\mathtt{x}) > 6$

formula:

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

Constraints:encoding into smtlib

```
(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: the formula encoded in smt-lib 2 format

constraints:

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

formula:

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

```
sat
((x "abcAAAA") (y "abcAAAA") (z "cAAAA") (l "AAAA"))
```

Figure: the output from cvc4

Introduction to CVC4

- CVC4
  - CVC4 is an automatic theorem prover for Satisfiability Modulo Theories.
  - along with other theories it also supports 'Theory of Strings'.
  - the theory solver for strings is implemented as natively.
- Features:
  - allows constraints with unbounded Strings,
  - does not translate the problem into other theories (e.g. bitvectors)
  - the procedure is algebraic in approach.

Overview of the procedure

- The procedure is defined as a set of derivation rules.
- The repeated application of rules produces a derivation tree, where each node in the derivation tree is called a *configuration*.
- while the rule application, the tree splits with new configuration, where
  - no further rule application is possible
  - or the configuration is `unsat`, then backtrack and take another branch if exists
- If the procedure ends up with a *closed* tree, then it concludes as `unsat`.
- or If the procedure ends up with a *saturated* configuration, then it concludes as `sat`.

Procedure

The main procedure is based on the repeated application of the rules according to the following steps,

1. *Check conflicts*
2. *Propagate*
3. *Split byLength*
4. *Normalize*
5. *Partition*
6. *Check cardinality*

Note:

- S : set of string constraints.
- A : set of arithmetic constraints.

Procedure: Step 0: *Start of the procedure*

- This is the start of the procedure.

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

- All the terms must be in normalized form, according to the above reduction rules.

  e.g. $\texttt{con}(''ab'', \texttt{con}(''c'', l)) \to \texttt{con}(''abc'', l)$
  e.g. $\texttt{len}(''abc'') \to 3$
  e.g. $\texttt{len}(\texttt{con}(''abc'', ''def'')) \to 6$

Procedure: Step 1: *Check conflicts*

- Check is there any contradiction exists in the current constraints
- Rules used: `S-Conflict` , `A-Conflict`.

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

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

- Examples:
  - e.g. $S : \{\cdots, x \approx \epsilon, x \not\approx \epsilon, \cdots\} \rightarrow \texttt{unsat}$
  - e.g. $S : \{\cdots, x \approx y, x \not\approx y, \cdots\} \rightarrow \texttt{unsat}$
  - e.g. $A : \{\texttt{len}(x) \approx \texttt{len}(y), \texttt{len}(x) \not\approx \texttt{len}(y)\} \rightarrow \texttt{unsat}$

Procedure: Step 2: *Propagate*

- Introduce new constraints induced by constraints of ot (e.g. $\mathcal{T}_{LIA}$ and $\mathcal{T}_{SL}$).
- Rules used : `S-Prop`, `A-Prop`.

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

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

- Examples:
  - e.g. $S \models (\texttt{len}\ x \approx \texttt{len}\ y) \rightarrow A := A, \texttt{len}\ x \approx \texttt{len}\ y$
  - e.g. $S \models (\texttt{len}\ x \approx \texttt{len}\ "abc") \rightarrow A := A, \texttt{len}\ x = 3$

Procedure: Step 3: *Split by Length*

- Introduce new constraint into the set of arithmetic constraints for equalities in string constraints.
- Introduce branching for free variables in string constraints.
- Rules used: `Len` and `Len-Split`.

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

- Examples:
  - e.g. $S \models (x \approx y) \rightarrow A := A, \texttt{len}\ x \approx \texttt{len}\ y$
  - e.g. $S \models (x \approx "abc") \rightarrow A := A, \texttt{len}\ x = 3$

Procedure: Step 4: *Normalize*

- Compute the normalized form for each term.
- If there is term not in normalized, apply splitting and then finally unify.
- Rules used: `S-Cycle`, `L-Split`, `F-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:
  - e.g. $\texttt{con}(x, m) \approx \texttt{con}(y, n), \texttt{len}(x) \approx \texttt{len}(y) \rightarrow S := S, x \approx y$

Procedure: Step 5: *Partition*

- Each equivalence class should be in their corresponding group (bucket).
- If the partition is not compete, apply splitting on the free variables.
- Rules used: `D-Base`, `D-Add`, `S-Split` and `L-Split`.

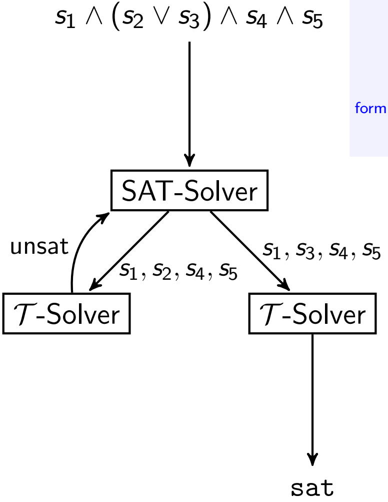
$$\texttt{S-Split}\,\frac{x, y \in \mathcal{V}(S) \quad x \approx y, x \not\approx y \in \mathcal{C}(S)}{S := S, x \approx y \parallel S := S, x \not\approx y}$$

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

Procedure: Step 6: *Check cardinality*

- For each bucket B introduce new an arithmetic constraint, as the alphabet $\mathcal{A}$ is finite.
- Performed as a last step of the procedure.
- For example,
  - If
    - we have 256 characters, and
    - S entails that 257 distinct strings of length 1 exist
  - Then
    - S is unsatisfiable

Derivation Tree Example



constraints:

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

formula:

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

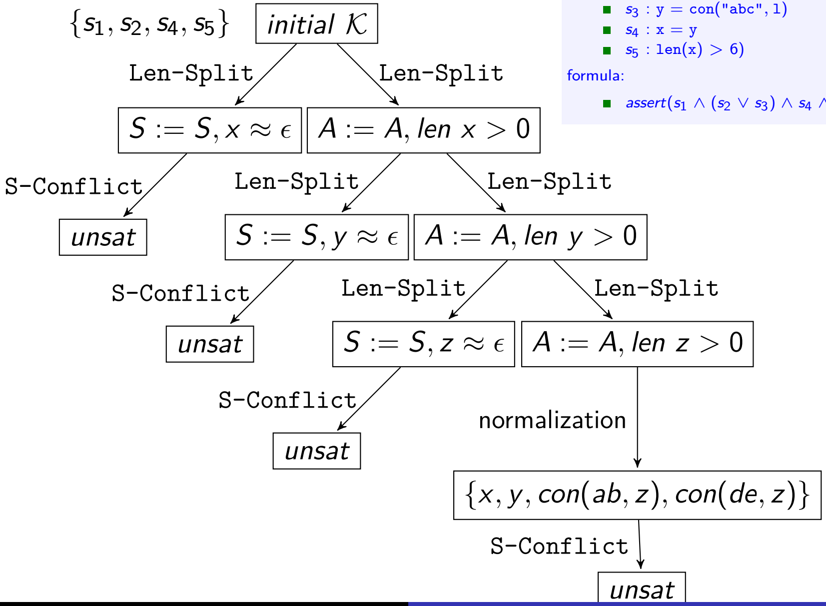
Left derivation tree

constraints:

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

formula:

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



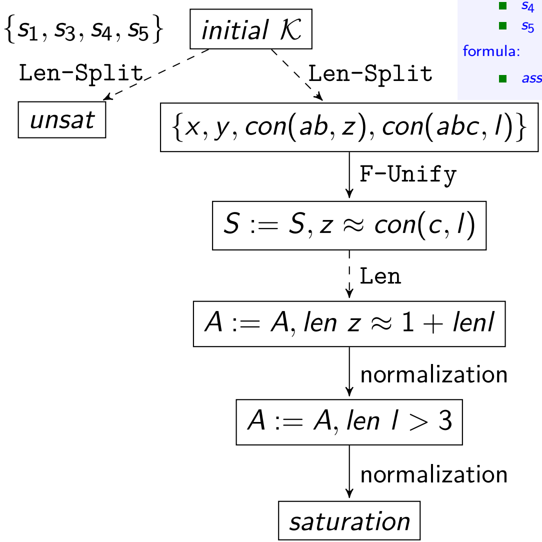
Right derivation tree

constraints:

- s_1 : $\mathtt{x} = \mathtt{con}(\mathtt{"ab"}, \mathtt{z})$
- s_2 : $\mathtt{y} = \mathtt{con}(\mathtt{"de"}, \mathtt{z})$
- s_3 : $\mathtt{y} = \mathtt{con}(\mathtt{"abc"}, \mathtt{l})$
- s_4 : $\mathtt{x} = \mathtt{y}$
- s_5 : $\mathtt{len}(\mathtt{x}) > 6)$

formula:

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



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

Correctness

- *Refutation Sound*: when the procedure answers with `unsat`, it can be trusted.
- *Solution Sound*: when the procedure answers with `sat`, it can be trusted.
- *Solution Complete*: eventually get a model by finite model finding.
- *Refutation Complete*: the procedure may <u>not</u> terminate for `unsat` problems.

Evaluation

- There was an evaluation conducted by the authors with:
  - Z3-STR.
  - Kaluza.
- 50K benchmarks generated by Kudzu were used.
- CVC4 string solver performed better.
- Since the string solver is natively integrated

Observation

- Positive:.
  - The idea of implementing the string solver natively is unique.
  - This approach allowed high interaction with core of cvc4.
  - Since it is not a plugin, the performance is better than others.
  - This approach allowed uses of general purpose Arithmetic solver.
- Negative:
  - The application of the derivation rules is complicated.
  - Limited expressiveness, only supported few string methods.