Solving String Constraints through Hardware/Software Model Checking

Jie-Hong R. Jiang[1] and Fang Yu[2]

1. Graduate Institute of Electronics Engineering
National Taiwan University, Taiwan
http://alcom.ee.ntu.edu.tw
2. Department of Management Information Systems
National Chengchi University, Taiwan
http://soslab.nccu.edu.tw

Meeting on String Constraints and Applications (MOSCA'19), May 6–9, 2019, Bertinoro, Italy

MOSCA, August 28, 2019

Input validation and sanitization is error-prone

- Programs that propagate and use malicious user inputs without validation and sanitization, or with improper validation and sanitization, are vulnerable to attacks such as Injections in Web applications.
- These string-related vulnerabilities are notorious and widely publicized [OWASP17].

OWASP Top 10 - 2013	→	OWASP Top 10 - 2017
A1 – Injection	→	A1:2017-Injection
A2 – Broken Authentication and Session Management	→	A2:2017-Broken Authentication
A3 – Cross-Site Scripting (XSS)	↘	A3:2017-Sensitive Data Exposure

String analysis techniques are needed

- It drives the need for automated tool support in analyzing string manipulating programs.
  - Hampi [Kiezun et al, ISSTA'09, Ganesh et al. CAV'11, TOSEM'12]
  - Z3str, Z3str2, Z3str3, and Z3strBV [Zheng et al. FSE'13, CAV'15], [Berzish et al. FMCAD'17], [Subramanian et al., ICSE'17]
  - CVC4 [Liang et al. CAV'14]
  - S3, and S3P [Trinh et al., CCS'14, CAV'16]
  - Norn and TRAU [Abdulla et al, CAV'14, CAV'15], [Abdulla et al, PLDI'17]
  - Sloth [Lin et al., POPL'16, Holik. et al., POPL'18]
  - Stranger and ABC [Yu et al, TACAS'10], [Aydin et al., CAV'15 and FSE'18]
  - Slog and **Slent** [Wang et al. CAV'16 and ASE'18]

Solving complex string constraints remains challenging

- String constraint solving can be particularly hard when the constraints involve complex string operations and involve both string and integer variables.
- Specifically, it has been shown that solving string constraints with *replace all* and length constraints is undecidable. [Chen et al. POPL'18]
- The *replace all* operation defines the replace of a match pattern with a replacement pattern for the sentence within a given set of language.
- It is widely used in input sanitization functions in Web applications.

A motivating example

Is the constraint satisfiable?

$$
\begin{aligned}
X_1 &\in a^*, \\
X_2 &\in b^*, \\
X_3 &= X_1.X_2, \\
X_4 &= \textsc{replace}(X_3, a^+b, ba), \\
\textsc{Len}(X_1) &= \textsc{Len}(X_2), \\
\textsc{Len}(X_1) &> \textsc{Len}(X_4).
\end{aligned}
$$

Is the constraint satisfiable?

- $(X_3 = X_1.X_2)$ and $(\textsc{Len}(X_1) = \textsc{Len}(X_2))$ ensure that X_3 is in the language $a^n b^n$, for $n \geq 0$ being the lengths of X_1 and X_2.
- X_4 is obtained by performing *language to language replacement* on X_3.
- For $X_4 = \textsc{replace}(X_3, a^+b, ba)$, a substring of the form $a^m b$, for some $1 \leq m \leq n$, in the middle of $a^n b^n$ will be replaced with ba.
- In this case, we have $\textsc{Len}(X_4) = 2n - (m+1) + 2 > n = \textsc{Len}(X_1)$, which contradicts the last constraint $\textsc{Len}(X_1) > \textsc{Len}(X_4)$.
- Hence the set of constraints is *unsatisfiable*.

SMT-based string constraint solving

- The SMT-based approaches, e.g., S3, Z3STR3, CVC4, Norn, for string constraint solving are native to deal with **length constraints**.
- While these DPLL(T)-based solvers handle a variety of string constraints, including word equations, regular expression membership, length constraints, and (more rarely) regular/rational relations; the solvers can not handle *replace-all* operation.
- The work [Trinh et al., CAV'16] that extends S3 to S3P addresses this issue with recurrence to reason such operations.

SMT-based string constraint solving

- $Y = \textsc{replace}(X, R_1, R_2)$ can be recursively defined:

$$
\begin{aligned}
&((Y = X) \wedge X \notin (\Sigma^*.R_1.\Sigma^*)) \vee \\
&((X = X_1.X_2.X_3) \wedge (X_1 \notin (\Sigma^*.R_1.\Sigma^*)) \wedge \\
&(X_2 \in R_1) \wedge (Y = X_1.Y_1.Y_2) \wedge (Y_1 \in R_2) \wedge \\
&(Y_2 = \textsc{replace}(X_3, R_1, R_2)),
\end{aligned}
$$

- However, the recursive operation may cause non-termination, and lead to non-robust results of constraint solving.

Automata-based string constraint solving

- For automata-based solvers, e.g., Stranger or ABC, the replacement operation can be naturally achieved by **automata-based construction**.
- However, the satisfying values of variables X_1, X_2, X_3, X_4 in the above example are not regular due to the condition imposed by the length constraints. They cannot be represented precisely with finite-state automata.
- The regular approximation on string and length relations leads imprecision.

The question is:

Can we take advantage on automata construction to model complex string operations but also deal with length constraints precisely?

The idea is:

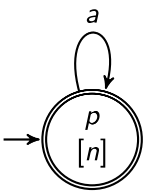
Attach an integer variable to track the length information of an automata.

- Such automata with length encoded integers are referred to as *length-encoded automata*.
- A non-epsilon transition of an automaton should incur a length increment by one, and thus the integer indicates the length of the string currently taken by the automaton
- By setting the initial value of an integer to zero, after taking an input sequence, the final value of the integer will be the length of this sequence.
- Accepting conditions on n can then be added to restrict the accepting language.

Length-encoded Automata

To accept a simple language $\{aaaa\}$:

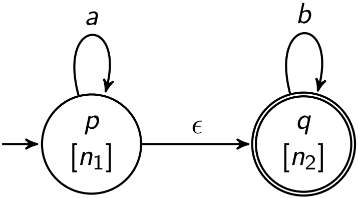
- Attach n to a finite automata A that accepts a^*.
- Add $n = 0$ to the initial state
- Add $n = 4$ to the accepting state



Length-encoded automata

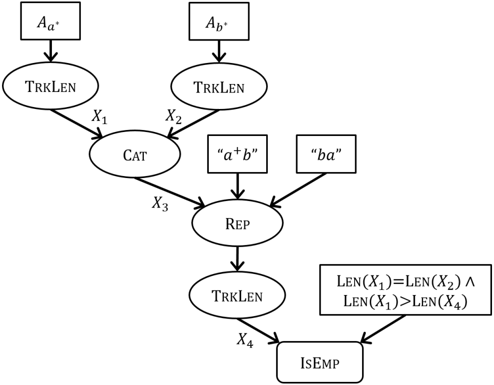
To accept the context free language $\{a^n b^n \mid n \in \mathbb{N}\}$:

- Concatenate two length encoded automata that recognize a^* and b^*, respectively.
- n_1 counts the number of a's taken so far on state p, and n_2 counts the number of b's taken so far on state q.
- Add $n_1 = 0$ and $n_2 = 0$ to the initial state and $n_1 = n_2$ to the accepting state.



Length-encoded automata

- To accept the language that satisfies the motivating example:



The constraint solving problem can be reduced to the language emptiness checking problem.

Language emptiness checking

To exploit software model checking algorithms to language emptiness checking:

- We first represent the *finite-state automaton* $A = (Q, \Sigma, I, O, T)$ with characteristic functions:
  - $I(\vec{s}) : Q \rightarrow \mathbb{B}$,
  - $T(\vec{x}, \vec{s}, \vec{s}') : \Sigma \times Q \times Q \rightarrow \mathbb{B}$, and
  - $O(\vec{s}) : Q \rightarrow \mathbb{B}$,
  - where $\vec{x}$, $\vec{s}$, and $\vec{s}'$ are the input, current-state, and next-state variables, respectively,

Language emptiness checking

To exploit software model checking algorithms to language emptiness checking:

- A (finite) *string* $\sigma_1, \ldots, \sigma_n$ is accepted if there exist states $q_1, \ldots, q_{n+1}$ such that
  - $I(q_1) = 1$ (for q_1 being an initial state),
  - $O(q_{n+1}) = 1$ (for q_{n+1} being an accepting state), and
  - the sequence q_1, σ_1, q_2, σ_2, $\ldots$, q_{n+1} satisfies $T(\sigma_i, q_i, q_{i+1})$ for $i = 1, \ldots, n$
- This can be done by iteratively expanding transition relations until that an accepting word has been found or a fixpoint has been reached.
- The process may not terminate when states are infinite.

Infinite-state automata construction

- We extend the characteristic functions of finite state automata to infinite state automata.
- Insert auxiliary (integer) state variables to track length information and restrict accepting languages
- We show how to construct corresponding characteristic functions through automata manipulations.
    - *length tracking*, *intersection*, *union*, *concatenation*, *deletion*, *replacement*, *reversion*, *prefix*, *suffix*, *substring*, and *index tracking*.

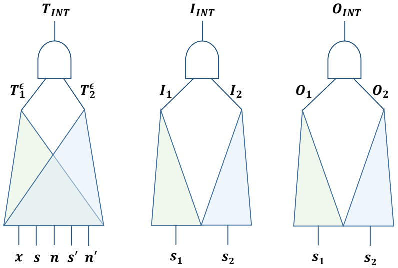
Length Tracking: $A^L = \textsc{TrkLen}(A)$

- Given a *finite automaton* A with its characteristic functions $T(\vec{x}, \vec{s}, \vec{s}')$, $I(\vec{s})$, and $O(\vec{s})$, $A^L = \textsc{TrkLen}(A)$, which embeds an integer variable n to count the number of transitions in T, can be constructed as:

$$\begin{aligned}
T^L(\vec{x}, \vec{s}, n, \vec{s}', n') &= T(\vec{x}, \vec{s}, \vec{s}') \wedge (((\vec{x} \neq \epsilon) \wedge (n' = n + 1)) \\
&\quad \vee ((\vec{x} = \epsilon) \wedge (n' = n))) \\
I^L(\vec{s}) &= I(\vec{s}) \\
O^L(\vec{s}) &= O(\vec{s})
\end{aligned}$$

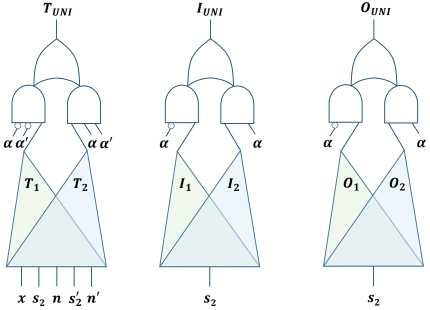
Intersection: $\mathcal{L}(A_{\mathrm{INT}}) = \mathcal{L}(A_1) \cap \mathcal{L}(A_2)$

- $\vec{s} = (\vec{s}_1, \vec{s}_2)$ and $\vec{n} = (\vec{n}_1, \vec{n}_2)$.
- T^{ϵ} denotes the transition relation derived from T with an additional ϵ self-transition added to each state.



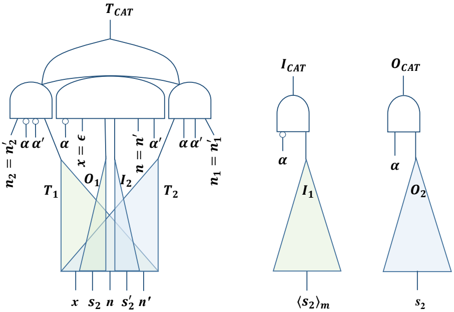
Union: $\mathcal{L}(A_{\mathrm{UNI}}) = \mathcal{L}(A_1) \cup \mathcal{L}(A_2)$

- Assume $|\vec{s}_1| \leq |\vec{s}_2|$. The state variables $\vec{s}_1$ of A_1 are merged into $\vec{s}_2$. $\vec{s} = (\vec{s}_2, \alpha)$, $\vec{n} = (\vec{n}_1, \vec{n}_2)$.
- An auxiliary bit α is used to distinguish states of A_1 (if α valuates to 0) or A_2 (if α valuates to 1).



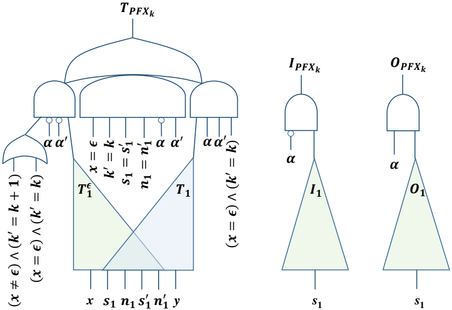
Concatenation: $\mathcal{L}(A_{\mathrm{CAT}}) = \mathcal{L}(A_1).\mathcal{L}(A_2)$

- Assume $|\vec{s}_1| \leq |\vec{s}_2|$. The state variables $\vec{s}_1$ of A_1 are merged into $\vec{s}_2$. $\vec{s} = (\vec{s}_2, \alpha)$ and $\vec{n} = (\vec{n}_1, \vec{n}_2)$.
- α is used to distinguish states on A_1 (if α valuates to 0) or on A_2 (if α valuates to 1).



Prefix: $\mathcal{L}(A_{\mathrm{PFX}_k}) = \{\vec{\sigma} | \exists \vec{\rho}.[\vec{\sigma}\vec{\rho} \in \mathcal{L}(A_1)] \wedge len(\vec{\sigma}) = k\}$

- $\vec{s} = (\vec{s_1}, \alpha)$ and $\vec{n} = (\vec{n_1}, k)$
- k is used to track $len(\vec{\sigma})$, and α is used to distinguish prefix states (if α valuates to 0) and tail states (if α valuates to 1).



Take Away

- Encode length information to string automata as length encoded automata
- Construct characteristic functions of length-encoded automata through automata manipulations that correspond to string and length constraints
- Leverage a symbolic model checker for infinite state systems as an engine for language emptiness checking

Tool Implementation and Settings

- The proposed method was implemented as a tool, called SLENT, using IC3IA [Cimtti et al. TACAS'14] as the backend symbolic model checker for emptiness checking on string and integer constraints.
- To evaluate the effectiveness of our tool, SLENT is compared against state-of-the-art mixed string and integer constraint solvers, including ABC, CVC4, NORN, S3P, TRAU, and Z3STR3.
- SLOTH does not support length constraint solving in the current released version and is excluded from the comparison.

Concatenation and length constraint solving

- RQ1: How SLENT performs compared to other solvers in solving pure concatenation and length constraints?
- 2000 test cases randomly sampled from the Kaluza benchmarks that involve only string concatenation operations and length constraints.

solver	time (s)	#SAT	#UNSAT	#TO
Z3STR3	56.46	1017	983	0
CVC4	88.89	1017	983	0
NORN	2025.30	1013	983	4
ABC	255.76	1013	983	4
S3P	137.90	1015	983	2
TRAU	123.85	1017	983	0
SLENT	1397.82	1013	983	4

String to string replace-all operation and length constraint solving

- RQ2: How SLENT performs compared to other solvers in solving string-to-string replacement, concatenation and length constraints?
- 236 test cases from the Stranger benchmarks with additional length constraints inserted.

solver	time(s)	#SAT	#UNSAT	#TO	#abort
ABC	2282.84	109(31)	111(0)	0	16
S3P	605.79	30(0)	114(3)	22	70
TRAU	687.49	54(2)	139(22)	5	38
SLENT	26692.55	88(0)	141(0)	7	0

Language to language replace-all operation and length constraint solving

- RQ3: How SLENT performs compared to other solvers in solving language-to-language replacement, concatenation and length constraints?
- 101 test cases from the Stranger benchmarks with additional length constraints inserted.

solver	time (s)	#SAT	#UNSAT	#TO	#abort
ABC	977.80	46(2)	41(0)	1	13
SLENT	4413.25	44(0)	38(0)	19	0

Conclusion

- We present a novel symbolic model checking approach for solving string and integer constraints based on length-encoded automata.
- Our solver SLENT is particularly suitable for solving complex string and integer constraints.
- As SLENT precisely maintains the relation among string and length variables, no approximation is required for constraint solving unlike other existing automata-based methods.
- The experiment shows the unique benefit of the proposed method on solving constraints with replace-all operation over string variables and with complex length relation.
- As SLENT relies on off-the-shelf model checkers, it benefits from model checker advancements. Its performance and practicality may be improved over time.

Thank you

SLENT is available at:
https://github.com/NTU-ALComLab/SLENT