Solving String Constraints through Hardware/Software Model Checking

Jie-Hong R. Jiang\textsuperscript{1} and Fang Yu\textsuperscript{2}

1. Graduate Institute of Electronics Engineering
   National Taiwan University, Taiwan
   \hspace{1cm} http://alcom.ee.ntu.edu.tw

2. Department of Management Information Systems
   National Chengchi University, Taiwan
   \hspace{1cm} 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].
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?

\[ X_1 \in a^*, \]
\[ X_2 \in b^*, \]
\[ X_3 = X_1.X_2, \]
\[ X_4 = \text{REPLACE}(X_3, a^+b, ba), \]
\[ \text{LEN}(X_1) = \text{LEN}(X_2), \]
\[ \text{LEN}(X_1) > \text{LEN}(X_4). \]
Is the constraint satisfiable?

- \((X_3 = X_1.X_2)\) and \((\text{LEN}(X_1) = \text{LEN}(X_2))\) 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 \(X_4 = \text{REPLACE}(X_3, a^+ b, ba)\), 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 \(\text{LEN}(X_4) = 2n - (m + 1) + 2 > n = \text{LEN}(X_1)\), which contradicts the last constraint \(\text{LEN}(X_1) > \text{LEN}(X_4)\).
- Hence the set of constraints is \emph{unsatisfiable}.
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.
Y = \texttt{REPLACE}(X, R_1, R_2) can be recursively defined:

\[
\begin{align*}
((Y = X) \land X \not\in (\Sigma^* . R_1 . \Sigma^*)) \lor \\
((X = X_1 . X_2 . X_3) \land (X_1 \not\in (\Sigma^* . R_1 . \Sigma^*)) \land \\
(X_2 \in R_1) \land (Y = X_1 . Y_1 . Y_2) \land (Y_1 \in R_2) \land \\
(Y_2 = \texttt{REPLACE}(X_3, R_1, R_2)),
\end{align*}
\]

However, the recursive operation may cause non-termination, and lead to non-robust results of 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 to 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:
  - \( I(\vec{s}) : Q \to \mathbb{B} \),
  - \( T(\vec{x}, \vec{s}, \vec{s}') : \Sigma \times Q \times Q \to \mathbb{B} \), and
  - \( O(\vec{s}) : Q \to \mathbb{B} \),
- where \( \vec{x} \), \( \vec{s} \), and \( \vec{s}' \) are the input, current-state, and next-state variables, respectively,
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 $q_1, \sigma_1, q_2, \sigma_2, \ldots, q_{n+1}$ 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: $A^L = \text{TRKLEN}(A)$

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

$$T^L(\vec{x}, \vec{s}, n, \vec{s}', n') = T(\vec{x}, \vec{s}, \vec{s}') \land (((\vec{x} \neq \epsilon) \land (n' = n + 1)) \lor ((\vec{x} = \epsilon) \land (n' = n)))$$

$$I^L(\vec{s}) = I(\vec{s})$$

$$O^L(\vec{s}) = O(\vec{s})$$
Intersection: $\mathcal{L}(A_{\text{INT}}) = \mathcal{L}(A_1) \cap \mathcal{L}(A_2)$

- $\vec{s} = (\vec{s}_1, \vec{s}_2)$ and $\vec{n} = (\vec{n}_1, \vec{n}_2)$.
- $T^\epsilon$ denotes the transition relation derived from $T$ with an additional $\epsilon$ self-transition added to each state.
Union: $\mathcal{L}(A_{UNI}) = \mathcal{L}(A_1) \cup \mathcal{L}(A_2)$

- 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 $\alpha$ is used to distinguish states of $A_1$ (if $\alpha$ valuates to 0) or $A_2$ (if $\alpha$ valuates to 1).
Concatenation: \( \mathcal{L}(A_{\text{CAT}}) = \mathcal{L}(A_1).\mathcal{L}(A_2) \)

- 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 \( \vec{n} = (\vec{n}_1, \vec{n}_2) \).
- \( \alpha \) is used to distinguish states on \( A_1 \) (if \( \alpha \) valuates to 0) or on \( A_2 \) (if \( \alpha \) valuates to 1).
Prefix: \( \mathcal{L}(A_{\text{PFX}_k}) = \{ \vec{\sigma} \mid \exists \vec{\rho}. [\vec{\sigma} \vec{\rho} \in \mathcal{L}(A_1)] \land \text{len}(\vec{\sigma}) = k \} \)

- \( \vec{\sigma} = (\vec{s}_1, \alpha) \) and \( \vec{n} = (\vec{n}_1, k) \)
- \( k \) is used to track \( \text{len}(\vec{\sigma}) \), and \( \alpha \) is used to distinguish prefix states (if \( \alpha \) valuates to 0) and tail states (if \( \alpha \) 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
The proposed method was implemented as a tool, called Slent, using IC3ia [Cimatti 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.

<table>
<thead>
<tr>
<th>solver</th>
<th>time (s)</th>
<th>#SAT</th>
<th>#UNSAT</th>
<th>#TO</th>
</tr>
</thead>
<tbody>
<tr>
<td>Z3STR3</td>
<td>56.46</td>
<td>1017</td>
<td>983</td>
<td>0</td>
</tr>
<tr>
<td>CVC4</td>
<td>88.89</td>
<td>1017</td>
<td>983</td>
<td>0</td>
</tr>
<tr>
<td>Norn</td>
<td>2025.30</td>
<td>1013</td>
<td>983</td>
<td>4</td>
</tr>
<tr>
<td>ABC</td>
<td>255.76</td>
<td>1013</td>
<td>983</td>
<td>4</td>
</tr>
<tr>
<td>S3P</td>
<td>137.90</td>
<td>1015</td>
<td>983</td>
<td>2</td>
</tr>
<tr>
<td>Trau</td>
<td>123.85</td>
<td>1017</td>
<td>983</td>
<td>0</td>
</tr>
<tr>
<td>Slent</td>
<td>1397.82</td>
<td>1013</td>
<td>983</td>
<td>4</td>
</tr>
</tbody>
</table>
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.

<table>
<thead>
<tr>
<th>solver</th>
<th>time(s)</th>
<th>#SAT</th>
<th>#UNSAT</th>
<th>#TO</th>
<th>#abort</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABC</td>
<td>2282.84</td>
<td>109(31)</td>
<td>111(0)</td>
<td>0</td>
<td>16</td>
</tr>
<tr>
<td>S3P</td>
<td>605.79</td>
<td>30(0)</td>
<td>114(3)</td>
<td>22</td>
<td>70</td>
</tr>
<tr>
<td>TRAU</td>
<td>687.49</td>
<td>54(2)</td>
<td>139(22)</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>Slent</td>
<td>26692.55</td>
<td>88(0)</td>
<td>141(0)</td>
<td>7</td>
<td></td>
</tr>
</tbody>
</table>
Language to language replace-all operation and length constraint solving

- RQ3: How \texttt{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.

<table>
<thead>
<tr>
<th>solver</th>
<th>time (s)</th>
<th>#SAT</th>
<th>#UNSAT</th>
<th>#TO</th>
<th>#abort</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABC</td>
<td>977.80</td>
<td>46(2)</td>
<td>41(0)</td>
<td>1</td>
<td>13</td>
</tr>
<tr>
<td>\texttt{SLent}</td>
<td>4413.25</td>
<td>44(0)</td>
<td>38(0)</td>
<td>19</td>
<td>0</td>
</tr>
</tbody>
</table>
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.
SLENT is available at:
https://github.com/NTU-ALComLab/SLENT