An Introduction to String Solvers

Shuanglong Kan

Automated Reasoning Group, TU Kaiserslautern

Table of contents

Why strings are important in Computer Systems

Strings in Programming Languages

Javascript code snipt

```
var x = goog.string.htmlEscape(name);
var y = goog.string.escapeString(x);
nameElem.innerHTML =
  '<button onclick= "viewPerson(\'' + y + '\')">' +
          x + '</button>';
```

- `htmlEscape`: Escapes double quote ‘"’ and single quote ‘'’ characters in addition to ‘&’, ‘<’, and ‘>’ so that a string can be included in an HTML tag attribute value within double or single quotes.
- `escapeString`: Takes a string and returns the escaped string for that input string

Strings in Programming Languages

A Python code snippet

```
# s1, s2: strings with delimiter '-'
for x in s1.split('-')
  for y in s2.split('-')
    assert(len(x) > len(y))
}
```

Cross Site Scripting (XSS) attack

XSS attack: The attacker aims to execute malicious scripts in a web browser of the victim by including malicious code in a legitimate web page or web application.

Cross Site Scripting (XSS) attack

XSS attack: The attacker aims to execute malicious scripts in a web browser of the victim by including malicious code in a legitimate web page or web application.

Server-side pseudocode:

```
print "<html>"
print "<h1>Most recent comment</h1>"
print database.latestComment
print "</html>"
```

Cross Site Scripting (XSS) attack

XSS attack: The attacker aims to execute malicious scripts in a web browser of the victim by including malicious code in a legitimate web page or web application.

Server-side pseudocode:

```
print "<html>"
print "<h1>Most recent comment</h1>"
print database.latestComment
print "</html>"
```

Malicious comments (injections)

```
<script>doSomethingEvil();</script>
```

Filtering XSS attacks by regular expressions

Malicious comments

```
<script>doSomethingEvil();</script>
```

Filtering XSS attacks by regular expressions

Malicious comments

```
<script>doSomethingEvil();</script>
```

Regular expressions
$.* < script > .* < /script > .*$

Filtering XSS attacks by regular expressions

Malicious comments

```
<script>doSomethingEvil();</script>
```

Regular expressions
$.* < script > .* < /script > .*$

We need to check
comment $\in \mathcal{L}(.* < script > .* < /script > .*)$

Amazon Cloud Access Control Policies

Amazon policy control examples:

```
((allow,
   principal : students,
   action : getObject,
   resource : cs240/Exam.pdf),
(allow,
    principal : tas,
    action : getObject,
    resource : (cs240/Exam.pdf, cs240/Answer.pdf)))
```

Amazon Cloud Access Control Policies

Amazon policy control examples:

```
(allow,
  principle: *,
  action: getObject,
  resource: cs240,
  condition : (StringEquals, aws:SourceVpc, vpc-111bbb222),
              (StringLike, s3:prefix, cs240/Exam*)
)
```

Amazon Cloud Access Control Policies

Amazon policy control examples:

```
(allow,
  principle: *,
  action: getObject,
  resource: cs240,
  condition : (StringEquals, aws:SourceVpc, vpc-111bbb222),
              (StringLike, s3:prefix, cs240/Exam*)
)
```

- "`*`" denotes any string (`.*`)
- "`StringEquals`" denotes the value of `aws:SourceVpc` is equal to `vpc-111bbb222`
- "`StringLike`" denotes membership constraints, that is, the value of `s3:prefix` is a filename under the directory `cs240/Exam*`. More formally, $\mathit{prefix} \in \mathcal{L}(cs240/\mathit{Exam}.*)$

Regex Denial-of-Service (ReDos) Attack

- ReDos is an algorithmic complexity attack

Regex Denial-of-Service (ReDos) Attack

- ReDos is an algorithmic complexity attack
- It produces a denial-of-service by providing a regular expression that takes a very long time to evaluate.

Regex Denial-of-Service (ReDos) Attack

- ReDos is an algorithmic complexity attack
- It produces a denial-of-service by providing a regular expression that takes a very long time to evaluate.
- The attack exploits the fact that most regular expression implementations have exponential time worst case complexity

Regex Denial-of-Service (ReDos) Attack

- ReDos is an algorithmic complexity attack
- It produces a denial-of-service by providing a regular expression that takes a very long time to evaluate.
- The attack exploits the fact that most regular expression implementations have exponential time worst case complexity
- An attacker can thus provide such a regular expression to make the server either slowing down or becoming unresponsive

Regex Denial-of-Service (ReDos) Attack

- ReDos is an algorithmic complexity attack
- It produces a denial-of-service by providing a regular expression that takes a very long time to evaluate.
- The attack exploits the fact that most regular expression implementations have exponential time worst case complexity
- An attacker can thus provide such a regular expression to make the server either slowing down or becoming unresponsive

The efficieny of processing regular expressions is very important

What is a String Solver

String Operations

- Regular expression operations: intersection, union, differences, membership

String Operations

- Regular expression operations: intersection, union, differences, membership
- String replace and replaceAll

String Operations

- Regular expression operations: intersection, union, differences, membership
- String replace and replaceAll
- String programs with integers.
- ......

What is the string constraint solving

The following is the grammar of the input language of a simple string constraint solver:

$$s ::= v = c \mid v = v_1 + v_2 \mid v = \text{replace}(v_1, r, v_2) \mid v \textit{ in } r \mid s_1; s_2$$

The symbol c denotes a constant string, like "Hello, world". The symbol r denotes a regular expression.

The concatenation function $v_1 + v_2$ concatenates the value of v_1 and the value of v_2.

The replace operation $replace(v_{str}, r_{pattern}, v_{replace})$ replace a substring in v_{str} that matches the $r_{pattern}$ with the string of $v_{replace}$.

For instance, $replace(\text{“Hello world!”}, H.*o, Hallo)$ is a new string “Hallo world!”

What is the string constraint solving

The following is the grammar of the input language of a simple string constraint solver:

$$s ::= v = c \mid v = v_1 + v_2 \mid v = \text{replace}(v_1, r, v_2) \mid v \textit{ in } r \mid s_1; s_2$$

The symbol c denotes a constant string, like "Hello, world". The symbol r denotes a regular expression.

The concatenation function $v_1 + v_2$ concatenates the value of v_1 and the value of v_2.

The replace operation $replace(v_{str}, r_{pattern}, v_{replace})$ replace a substring in v_{str} that matches the $r_{pattern}$ with the string of $v_{replace}$.

For instance, $replace($"Hello world!"$, H. * o, Hallo)$ is a new string "Hallo world!"

$replace($"Hello world!"$, H. * o, Hallo)$ can also be another new string "Hallorld!"

What is the string constraint solving: An example

An example of string constraints:

```
v3 = v2 + v1;
v4 = replace(v3, v5, v6);
v5 in (.*)01(.*);
v6 = "111";
v3 in (0*)11(0*)
```

The string constraint solver should answer the question: If there exists the values (constant strings) for the variables $v_1, v_2, v_3, v_4, v_5, v_6$ such that all the equations and regular membership constraints are satisfied.

Model Checking Regular Language Constraints

Model Checking Regular Language Constraints[1]

- How to solve regular language emptiness checking problem
- Mapping language emptiness checking problem to hardware model checking problem using IC3.

[1] A work by Arlen Cox and Jason Leasure

Hardware model checking using IC3

Definition (Transition systems)
A finite transition system is described by a pair of propositional logic formulas:

- an initial condition $I(\bar{x})$ and
- a transition relation $T(\bar{i}, \bar{x}, \bar{x}')$

$\bar{x}$ denotes a sequence of boolean variables $x_1, x_2, \ldots, x_n$

$\bar{i}$ denotes a sequence of boolean variables $i_1, i_2, \ldots, i_n$

$I(\bar{x})$ is a boolean formula over the variables $\bar{x}$.

$T(\bar{i}, \bar{x}, \bar{x}')$ is a boolean formula over the variables $\bar{x}$, $\bar{x}'$, and $\bar{i}$

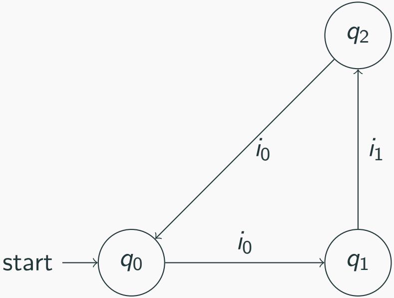


Figure 1: An example of transition systems

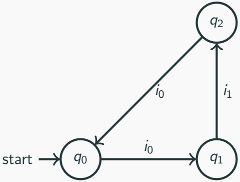
Table 1: Encoding of states

q_0	0	00	$\neg x_1 \wedge \neg x_0$
q_1	1	01	$\neg x_1 \wedge x_0$
q_2	2	10	$x_1 \wedge \neg x_0$

Table 2: Encoding of labels

i_0	0	$\neg i$
q_1	1	i

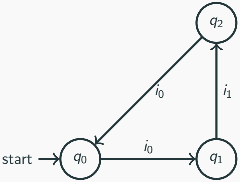
Hardware model checking using IC3



q_0	0	00	$\neg x_1 \land \neg x_0$
q_1	1	01	$\neg x_1 \land x_0$
q_2	2	10	$x_1 \land \neg x_0$

i_0	0	$\neg i$
q_1	1	i

Hardware model checking using IC3



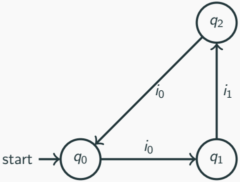
q_0	0	00	$\neg x_1 \wedge \neg x_0$
q_1	1	01	$\neg x_1 \wedge x_0$
q_2	2	10	$x_1 \wedge \neg x_0$

i_0	0	$\neg i$
q_1	1	i

The encoding of transition labels:

- (q_0, i_0, q_1) is encoded as: $t_1 \triangleq \underbrace{\neg x_1 \wedge \neg x_0}_{q_0} \wedge \underbrace{\neg i}_{i_0} \wedge \underbrace{\neg x_1' \wedge x_0'}_{q_1}$
- (q_1, i_1, q_2) is encoded as: $t_2 \triangleq \underbrace{\neg x_1 \wedge x_0}_{q_1} \wedge \underbrace{i}_{i_1} \wedge \underbrace{x_1' \wedge \neg x_0'}_{q_2}$
- (q_2, i_1, q_0) is encoded as: $t_3 \triangleq \underbrace{x_1 \wedge \neg x_0}_{q_2} \wedge \underbrace{\neg i}_{i_0} \wedge \underbrace{\neg x_1' \wedge \neg x_0'}_{q_0}$

Hardware model checking using IC3



q_0	0	00	$\neg x_1 \wedge \neg x_0$
q_1	1	01	$\neg x_1 \wedge x_0$
q_2	2	10	$x_1 \wedge \neg x_0$

i_0	0	$\neg i$
q_1	1	i

The encoding of transition labels:

- (q_0, i_0, q_1) is encoded as: $t_1 \triangleq \underbrace{\neg x_1 \wedge \neg x_0}_{q_0} \wedge \underbrace{\neg i}_{i_0} \wedge \underbrace{\neg x_1' \wedge x_0'}_{q_1}$
- (q_1, i_1, q_2) is encoded as: $t_2 \triangleq \underbrace{\neg x_1 \wedge x_0}_{q_1} \wedge \underbrace{i}_{i_1} \wedge \underbrace{x_1' \wedge \neg x_0'}_{q_2}$
- (q_2, i_1, q_0) is encoded as: $t_3 \triangleq \underbrace{x_1 \wedge \neg x_0}_{q_2} \wedge \underbrace{\neg i}_{i_0} \wedge \underbrace{\neg x_1' \wedge \neg x_0'}_{q_0}$

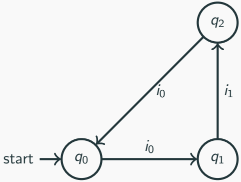
- $I(\bar{x}) = \neg x_1 \wedge \neg x_2$
- $T(\bar{i}, \bar{x}, \bar{x}') = t_1 \vee t_2 \vee t_3$

Hardware model checking using IC3

The safety property of model checking is a set of “bad” states that the system should not reach. We use a logic formula over state variables to specify the safety property.

Hardware model checking using IC3

The safety property of model checking is a set of "bad" states that the system should not reach. We use a logic formula over state variables to specify the safety property.



Assume q_2 is a bad state, the safety property is $P(\bar{x}) = x_1 \wedge \neg x_0$

Hardware model checking using IC3

The input of IC3 model checker is: $I(\bar{x}), T(\bar{i}, \bar{x}, \bar{x}'), P(\bar{x})$.

Hardware model checking using IC3

The input of IC3 model checker is: $I(\bar{x}), T(\bar{i}, \bar{x}, \bar{x}'), P(\bar{x})$.

The output of IC3 model checker is:

- sat, the bad states are not reachable from initial states
- unsat, there exists a bad state that are reachable from the initial states.

Hardware model checking using IC3

Interested in the algorithm of IC3?

Hardware model checking using IC3

Interested in the algorithm of IC3?

Read the paper: "Aaron R. Bradley: SAT-Based Model Checking without Unrolling. VMCAI 2011: 70-87"

Regular Expression (RE) Membership Checking

The grammar of membership checking language

$$RL ::= x \in L \mid \neg RL \mid RL \wedge RL \mid RL \vee RL$$

Regular Expression (RE) Membership Checking

The grammar of membership checking language

$$RL ::= x \in L \mid \neg RL \mid RL \wedge RL \mid RL \vee RL$$

Definition (NFA)
An NFA is 5-tuple $A = (\Sigma, Q, I, F, \Delta)$, where

- Σ is the alphabet,
- Q is a finite set of states,
- $I \subseteq Q$ is the set of initial states,
- $F \subseteq Q$ is the set of accepting states,
- $\Delta \subseteq Q \times \Sigma \times Q$ is the transition relations.

The drawbacks of using NFAs for RE

State space explosion problem in NFA construction from regular expressions, the intersection of two NFAs, and the complement of an NFA.

Especially the complement of an NFA performed via the powerset construction, which can result in an *exponential increase* in the number of states.

Alternating Finite Automata (AFA)

Definition (AFA)
An AFA is a tuple $M = (\Sigma, Q, q_0, F, \Delta)$, where Σ is an alphabet, Q is a finite set of states, $q_0 \in \mathcal{B}^+(Q)$ represents M's initial state, $F \subseteq Q$ is a set of accepting states, and $\Delta \subseteq Q \times \Sigma \to \mathcal{B}^+(Q)$.

$\mathcal{B}^+(Q)$ denotes the set of positive Boolean formulae over Q, that is the set of Boolean formulae built from *true*, *false*, and the members of Q using the binary connectives $\wedge$ and $\vee$.

The operations over AFAs

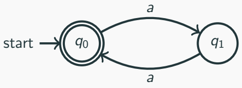
Let $M_1 = (\Sigma, Q_1, I_1, F_1, \Delta_1)$ and $M_2 = (\Sigma, Q_2, I_2, F_2, \Delta_2)$ be two AFAs.

- The intersection of M_1 and M_2 is $(\Sigma, Q_1 \cup Q_2, I_1 \wedge I_2, F_1 \cup F_2, \Delta_1 \cup \Delta_2)$
- The union of M_1 and M_2 is $(\Sigma, Q_1 \cup Q_2, I_1 \vee I_2, F_1 \cup F_2, \Delta_1 \cup \Delta_2)$
- The complement of M_1 is $(\Sigma, Q_1, \bar{I_1}, Q_1 \backslash F_1, \{(x, a) \mapsto \bar{p} : \Delta_1(x, a) = p\})$. $\bar{p}$ replaces every $\wedge$ with $\vee$ (and vice versa).

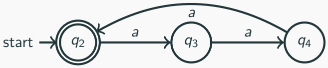
The operations over AFAs

Let $M_1 = (\Sigma, Q_1, I_1, F_1, \Delta_1)$ and $M_2 = (\Sigma, Q_2, I_2, F_2, \Delta_2)$ be two AFAs.

- The intersection of M_1 and M_2 is $(\Sigma, Q_1 \cup Q_2, I_1 \wedge I_2, F_1 \cup F_2, \Delta_1 \cup \Delta_2)$
- The union of M_1 and M_2 is $(\Sigma, Q_1 \cup Q_2, I_1 \vee I_2, F_1 \cup F_2, \Delta_1 \cup \Delta_2)$
- The complement of M_1 is $(\Sigma, Q_1, \bar{I}_1, Q_1 \backslash F_1, \{(x, a) \mapsto \bar{p} : \Delta_1(x, a) = p\})$. $\bar{p}$ replaces every $\wedge$ with $\vee$ (and vice versa).



M_1



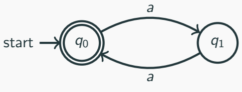
M_2

$M_1 = (\{a\}, \{q_0, q_1\}, q_0, \{q_0\}, \{(q_0, a, q_1), (q_1, a, q_0)\})$
$M_2 = (\{a\}, \{q_2, q_3, q_4\}, q_2, \{q_2\}, \{(q_2, a, q_3), (q_3, a, q_4), (q_4, a, q_2)\})$

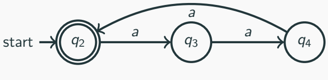
The operations over AFAs

Let $M_1 = (\Sigma, Q_1, I_1, F_1, \Delta_1)$ and $M_2 = (\Sigma, Q_2, I_2, F_2, \Delta_2)$ be two AFAs.

- The intersection of M_1 and M_2 is $(\Sigma, Q_1 \cup Q_2, I_1 \wedge I_2, F_1 \cup F_2, \Delta_1 \cup \Delta_2)$
- The union of M_1 and M_2 is $(\Sigma, Q_1 \cup Q_2, I_1 \vee I_2, F_1 \cup F_2, \Delta_1 \cup \Delta_2)$
- The complement of M_1 is $(\Sigma, Q_1, \bar{I}_1, Q_1 \backslash F_1, \{(x, a) \mapsto \bar{p} : \Delta_1(x, a) = p\})$. $\bar{p}$ replaces every $\wedge$ with $\vee$ (and vice versa).



M_1

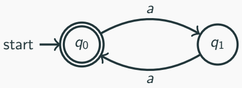


M_2

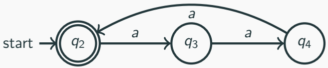
$M_1 = (\{a\}, \{q_0, q_1\}, q_0, \{q_0\}, \{(q_0, a, q_1), (q_1, a, q_0)\})$
$M_2 = (\{a\}, \{q_2, q_3, q_4\}, q_2, \{q_2\}, \{(q_2, a, q_3), (q_3, a, q_4), (q_4, a, q_2)\})$

The intersection of M_1 and M_2 is: $(\{a\}, \{q_0, q_1, q_2, q_3, q_4\}, q_0 \wedge q_2, \{q_0, a, q_2\}, \{(q_0, a, q_1), (q_1, a, q_0), (q_2, a, q_3), (q_3, a, q_4), (q_4, a, q_2)\})$

The operations over AFAs



M_1

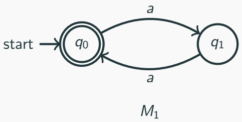


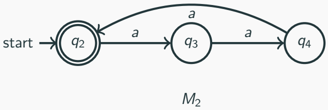
M_2

$M_1 = (\{a\}, \{q_0, q_1\}, q_0, \{q_0\}, \{(q_0, a, q_1), (q_1, a, q_0)\})$
$M_2 = (\{a\}, \{q_2, q_3, q_4\}, q_2, \{q_2\}, \{(q_2, a, q_3), (q_3, a, q_4), (q_4, a, q_2)\})$

The intersection of M_1 and M_2 is: $(\{a\}, \{q_0, q_1, q_2, q_3, q_4\}, q_0 \wedge q_2, \{q_0, a, q_2\}, \{(q_0, a, q_1), (q_1, a, q_0), (q_2, a, q_3), (q_3, a, q_4), (q_4, a, q_2)\})$

The operations over AFAs





$M_1 = (\{a\}, \{q_0, q_1\}, q_0, \{q_0\}, \{(q_0, a, q_1), (q_1, a, q_0)\})$
$M_2 = (\{a\}, \{q_2, q_3, q_4\}, q_2, \{q_2\}, \{(q_2, a, q_3), (q_3, a, q_4), (q_4, a, q_2)\})$

The intersection of M_1 and M_2 is: $(\{a\}, \{q_0, q_1, q_2, q_3, q_4\}, q_0 \wedge q_2, \{q_0, a, q_2\}, \{(q_0, a, q_1), (q_1, a, q_0), (q_2, a, q_3), (q_3, a, q_4), (q_4, a, q_2)\})$

An accepting run of the intersection of M_1 and M_2 is the following:

$$(q_0 \wedge q_2) \xrightarrow{a} (q_1, q_3) \xrightarrow{a} (q_0, q_4) \xrightarrow{a} (q_1, q_2) \xrightarrow{a} (q_0, q_3) \xrightarrow{a} (q_1, q_4) \xrightarrow{a} (q_0, q_2).$$

This run accepts the word "*aaaaaa*"

The Translation from AFAs to IC3 inputs

Let $M = (\Sigma, Q, I, F, \Delta)$ be an AFA.

- for each $q \in Q$, its binary encoding is denoted as $B[q]$
- for each $a \in \Sigma$, its binary encoding is denoted as $B[a]$

The Translation from AFAs to IC3 inputs

Let $M = (\Sigma, Q, I, F, \Delta)$ be an AFA.

- for each $q \in Q$, its binary encoding is denoted as $B[q]$
- for each $a \in \Sigma$, its binary encoding is denoted as $B[a]$

The translation is the following:

- $I(\bar{x})$ translated from M is: $I[q_i \mapsto B[q_i]]$, for all $q_i \in Q$
- $T(\bar{x}, \bar{i}, \bar{x}')$ translated from M is:

$$\bigwedge_{q_i \in Q} B[q_i] = \left(\bigvee_{a \in \Sigma} B[a] \wedge \Delta(q_i, a)[q_j \mapsto B[q_j]] \right)$$

- $P(\bar{x})$ translated from M is: $\left(\bigvee_{q_i \in F} B[q_i] \right)$

Experiments

This method is compared with 3 tools:

- BRICS [3]
- DPRLE solver [2]
- Norn [1]

The first benchmark:

- regular expressions from http://regexlib.com
- language intersection
- language difference

Experiments

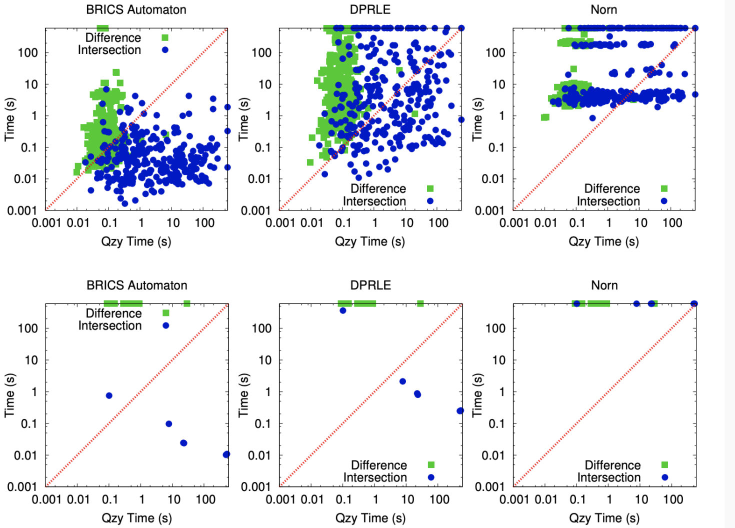


Fig. 6. Time to solve problems from RegExLib. First row are syntactcially large problems. Second row are difficult to determinize. Intersection problems are $x \in L_1 \wedge x \in L2$. Difference problems are $x \in L_1 \wedge x \notin L_2$.

Experiments

The second benchmark: increasing n in the following regular expressions.

- satisfiable difference
  $(x \in \hat{} \cdot [01]^* \cdot 1 \cdot [01]\{n\} \cdot \$ \wedge x \notin \hat{} \cdot [01] * \cdot 0 \cdot [01]\{n-1\} \cdot \$)$,
- unsatisfiable difference
  $(x \in \hat{} \cdot [01]^* \cdot 1 \cdot 1 \cdot [01]\{n\} \cdot \$ \wedge x \notin \hat{} \cdot [01] * \cdot 1 \cdot [01]\{n+1\} \cdot \$)$,
- satisfiable intersection
  $(x \in \hat{} \cdot [01]^* \cdot 1 \cdot [01]\{n\} \cdot \$ \wedge x \in \hat{} \cdot [01] * \cdot 0 \cdot [01]\{n-1\} \cdot \$)$,
- unsatisfiable intersection
  $(x \in \hat{} \cdot [01]^* \cdot 1 \cdot [01]\{n\} \cdot \$ \wedge x \in \hat{} \cdot [01] * \cdot 0 \cdot [01]\{n\} \cdot \$)$.

Experiments

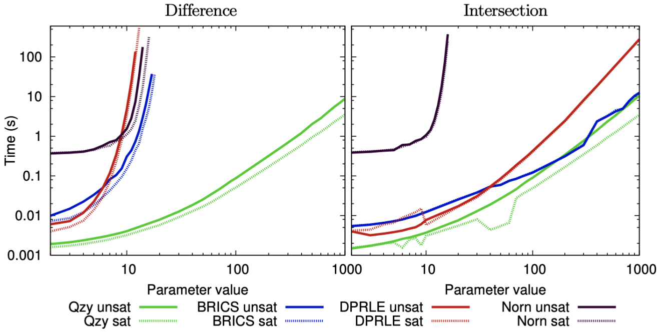


Fig. 7. Time to solve problems involving two parametric regular expressions. The complexity of the problem is determined by a single parameter.

Paper List for String Solvers

Paper List for String Solvers

Interested in more efficient algorithms solving regular constraints, like $x \in R$?

- Caleb Stanford, Margus Veanes, Nikolaj Bjørner: Symbolic Boolean derivatives for efficiently solving extended regular expression constraints. 620-635
- Loris D'Antoni, Zachary Kincaid, Fang Wang: A Symbolic Decision Procedure for Symbolic Alternating Finite Automata. MFPS 2018: 79-99

Paper List for String Solvers

Interested in the implementation of other string operations?

- Taolue Chen, Matthew Hague, Anthony W. Lin, Philipp Rümmer, Zhilin Wu. Decision procedures for path feasibility of string-manipulating programs with complex operations. Proc. ACM Program. Lang. 3(POPL): 49:1-49:30 (2019)
- Blake Loring, Duncan Mitchell, Johannes Kinder. Sound regular expression semantics for dynamic symbolic execution of JavaScript. 425-438.

Paper List for String Solvers

Interested in the decision procedure mixed with other data types?

- Taolue Chen, Matthew Hague, Jinlong He, Denghang Hu, Anthony Widjaja Lin, Philipp Rümmer, Zhilin Wu. A Decision Procedure for Path Feasibility of String Manipulating Programs with Integer Data Type. ATVA 2020: 325-342. **selected**
- Parosh Aziz Abdulla, Mohamed Faouzi Atig, Yu-Fang Chen, Bui Phi Diep, Julian Dolby, Petr Janku, Hsin-Hung Lin, Lukás Holík, Wei-Cheng Wu: Efficient handling of string-number conversion. 943-957.

Paper List for String Solvers

Interested in the application of string solvers in computer security?

- F. Yu, M. Alkhalaf, T. Bultan, and O. H. Ibarra. Automata-based symbolic string analysis for vulnerability detection. Formal Methods Syst. Des., 44(1):44–70, 2014.
- M. Trinh, D. Chu, and J. Jaffar. S3: A symbolic string solver for vulnerability detection in web applications. In G. Ahn, M. Yung, and N. Li, editors, Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, Scottsdale, AZ, USA, November 3-7, 2014, pages 1232–1243. ACM, 2014.

Thanks! Q & A

References

[1] Parosh Aziz Abdulla, Mohamed Faouzi Atig, Yu-Fang Chen, Lukás Holík, Ahmed Rezine, Philipp Rümmer, and Jari Stenman. Norn: An SMT solver for string constraints. In Daniel Kroening and Corina S. Pasareanu, editors, *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part I*, volume 9206 of *Lecture Notes in Computer Science*, pages 462–469. Springer, 2015.

References ii

[2] Pieter Hooimeijer and Westley Weimer. A decision procedure for subset constraints over regular languages. In Michael Hind and Amer Diwan, editors, *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2009, Dublin, Ireland, June 15-21, 2009*, pages 188–198. ACM, 2009.

[3] Anders Møller. dk.brics.automaton - finite-state automata and regular expressions for java. 2010. URL `https://cs.au.dk/~amoeller/`.