Decision Procedures for String Constraints

Ph.D. Proposal
Pieter Hooimeijer

University of Virginia
Motivation

Mitre Corp. data reported on http://www.attrition.org/
Mitre Corp. data reported on http://www.attrition.org/
Motivation

“String values have lost their innocence and are being used in many unforeseen contexts.”

[Thiemann05]
Motivation

“String values are being used for reasons other than their intended use.”

Report Frequency

- XSS
- Buffer
- SQL
- PHP Include

Year

Relative Frequency (%)
“String values have lost their innocence and are being used in many unforeseen contexts.”

[Thiemann05]
String values have lost their innocence and are being used in many unforeseen contexts.”

[Thiemann05]

Motivation

Now what?
Make string analysis available to a wider class of program analysis tools.
Outline

• String Constraint Solving
• Preliminary Results
• Proposed Research
Outline

• String Constraint Solving
  – example code
  – definitions
• Preliminary Results
• Proposed Research
Outline

- String Constraint Solving
  - example code
  - definitions
- Preliminary Results
- Proposed Research
Example

// v1 and v2 are user inputs

if (!(ereg('o(pp)+', v1))) {exit;}  
if (!(ereg('p*q', v2))) {exit;}

v3 = v1 . v2; // concat
if (v3 != 'oppppppqq') {exit;}
magic();
Query:
Will this code ever execute *magic*?
// v1 and v2 are user inputs

if (!ereg('^o(pp)+', v1)) {exit;}
if (!ereg('^p*q', v2)) {exit;}

v3 = v1 . v2; // concat
if (v3 != 'oppppppq') {exit;}

magic();
Outline

- String Constraint Solving
  - example code
  - definitions

- Preliminary Results

- Proposed Research
Outline

• String Constraint Solving
  – example code
  – definitions

• Preliminary Results

• Proposed Research
## Definitions

### String Constraint

<table>
<thead>
<tr>
<th>C ::=</th>
<th>E ∈ R</th>
<th>E ::=</th>
<th>V</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>E ∉ R</td>
<td></td>
</tr>
</tbody>
</table>

R: regex  
V: variable
Definitions

Constraint System

\[ S = \{ C_1, \ldots, C_n \} \]

where each \( C_i \in S \) is a well-formed string constraint.
Definitions

Decision Procedure

\[ D : \text{constraint system} \rightarrow \{ \text{Satisfiable, Unsatisfiable} \} \]
Definitions

Soundness

\[ D(S) = Sat. \] \rightarrow S \text{ is sat.} \]

Completeness

S \text{ is sat.} \rightarrow [D(S) = Sat.]
Definitions

Soundness

\[ D(S) = Sat. \] \rightarrow S \text{ is sat.} \]

Completeness

S \text{ is sat.} \rightarrow [D(S) = Sat.]
Definitions

Constraint System

\[ S = \{ C_1, \ldots, C_n \} \]

where each \( C_i \in S \) is a well-formed string constraint.

String Constraint

\[ C ::= E \in R \quad E ::= V \]
\[ \quad \mid E \notin R \quad \mid E \circ V \]

\[ R : \text{regex} \quad V : \text{variable} \]

Decision Procedure

\[ D : \text{constraint system} \rightarrow \{ \text{Satisfiable, Unsatisfiable} \} \]

Soundness

\[ [D(S) = \text{Sat.}] \rightarrow S \text{ is sat.} \]

Completeness

\[ S \text{ is sat.} \rightarrow [D(S) = \text{Sat.}] \]
Definitions

Constraint System

\[ S = \{ C_1, \ldots, C_n \} \]

where each \( C_i \in S \) is a well-formed string constraint.

String Constraint

\[
C ::= \ E \in R \quad E ::= \ V \\
\quad | \quad E \notin R \\
\quad | \quad E \circ V
\]

\( R : \text{regex} \quad V : \text{variable} \)

Decision Procedure

\[ D : \text{constraint system} \to \{ \text{Satisfiable}, \text{Unsatisfiable} \} \]

Soundness

\[ [D(S) = \text{Sat.}] \to S \text{ is sat.} \]

Completeness

\[ S \text{ is sat.} \to [D(S) = \text{Sat.}] \]
Definitions

Constraint System
\[ S = \{ C_1, \ldots, C_n \} \]
where each \( C_i \in S \) is a well-formed string constraint.

Decision Procedure
\[ D : \text{constraint system} \rightarrow \{ \text{Satisfiable}, \text{Unsatisfiable} \} \]

String Constraint
\[ C ::= \ E \in R \quad E ::= \ V \]
\[ \quad \ | \quad E \notin R \quad \ | \quad E \circ V \]
\[ R : \text{regex} \quad V : \text{variable} \]

Soundness
\[ [D(S) = \text{Sat.}] \rightarrow S \text{ is sat.} \]

Completeness
\[ S \text{ is sat.} \rightarrow [D(S) = \text{Sat.}] \]
Outline

• String Constraint Solving
  – example code
  – definitions

• Preliminary Results

• Proposed Research
## Existing Tools

<table>
<thead>
<tr>
<th>Tool</th>
<th>Encoding Format</th>
</tr>
</thead>
<tbody>
<tr>
<td>DPRLE [PLDI09]</td>
<td>Automata</td>
</tr>
<tr>
<td>Hampi [ISSTA09]</td>
<td>Encode to STP</td>
</tr>
<tr>
<td>Rex [ICST10]</td>
<td>Encode to Z3</td>
</tr>
<tr>
<td>Kaluza [Oakland10]</td>
<td>Encode to Hampi &amp; STP</td>
</tr>
<tr>
<td>Our Prototype</td>
<td>Lazy Automata</td>
</tr>
</tbody>
</table>
Questions

Make string analysis available to a wider class of program analysis tools.
Questions

• What is acceptable performance?

• What type of constraints should we allow?
Outline

• String Constraint Solving

• Preliminary Results
  – scalability
  – expressive utility

• Proposed Research
Scalability

Subjects:
- Decision Procedure for Regular Language Equations [PLDI09]
- Hampi [ISSTA09]
- Lazy Prototype
Scalability

Task: find a string that is in both

\([a-c]^*a[a-c]^{n+1}\)

and

\([a-c]^*b[a-c]^n\)
Scalability

Time to Generate First String

- DPRLE
- Hampi
- Our Prototype
Scalability

Time to Generate First String

Graph showing the time to generate first string with different symbols for Hampi and Our Prototype.
Scalability

• Existing approaches are less scalable than they could be on the tested benchmarks

• Interaction with an underlying solver introduces performance artifacts
Outline

• String Constraint Solving

• Preliminary Results
  – scalability
  – expressive utility

• Proposed Research
Expressive Utility

• Picked 88 PHP projects on SourceForge = 9.6 million LOC

• Tally: 111 distinct string functions
Expressive Utility

- **Index**: substr, strlen, strpos (total 63003)
- **Regular Expr**: preg_match, preg_replace (total 29141)
- **Format**: sprintf (17384)
- **Character**: chr, ord (35161)
- **Other**: str_replace, trim, htmlspecialchars, explode, implode (total 106628)
Expressive Utility

- **Index**: 63,003
  - (substr, strlen, strpos, ...)

- **Regex**: 29,141
  - (preg_match, preg_replace, ...)

Index: 63,003
Regex: 29,141
Expressive Utility

- Existing approaches typically support 'Regex,' but not 'Index' operations

- 'Index' operations were 2x as common in the sample under study
Outline

• String Constraint Solving
  • Preliminary Results
    – scalability
    – expressive utility
  • Proposed Research
Outline

• String Constraint Solving
• Preliminary Results

• Proposed Research
  – subset constraints
  – scalability through laziness
  – integer index operations
  – proof strategies
It is possible to construct a practical algorithm that decides the satisfiability of constraints that cover both string and integer index operations, scales up to real-world program analysis problems, and admits a machine-checkable proof of correctness.
It is possible to construct a practical algorithm that decides the satisfiability of constraints that cover both string and integer index operations, scales up to real-world program analysis problems, and admits a machine-checkable proof of correctness.
Outline

• String Constraint Solving
• Preliminary Results

• Proposed Research
  – subset constraints
  – scalability through laziness
  – integer index operations
  – proof strategies
Subset Constraints [PLDI'09]

\[
S ::= E \subseteq C \\
E ::= E \circ E \\
C :: V
\]
Approach

Input

1. $v_1 \subseteq c_1$
2. $v_2 \subseteq c_2$
3. $v_1 \circ v_2 \subseteq c_3$
Approach

Input

Cross Product

\((c_1 \circ c_2) \cap c_3\)

Sat.

Unsat.
Example

// v1 and v2 are user inputs

if (!ereg('o(pp)+', v1)) {exit;}
if (!ereg('p*q', v2)) {exit;}

v3 = v1 . v2; // concat
if (v3 != 'opppppq') {exit;}
magic();
Solution I: \( v_1 = \{ \text{opp} \} \)
\( v_2 = \{ \text{ppq} \} \)
Solution I: $v_1 = \{ \text{opp} \}$
$v_2 = \{ \text{ppq} \}$

Solution II: $v_1 = \{ \text{opppp} \}$
$v_2 = \{ \text{q} \}$
Algorithms and a Proof

• **Concat-Intersect (CI) algorithm:**
  – two variables, three constants; fixed form
  – mechanically verified proof in Coq 8.1pl3
  – proof size is ~1300 lines

• **Regular Matching Assignments (RMA):**
  – implemented in a tool, DPRPLE
  – applies CI procedure inductively
Evaluation

• Find SQL injection vulnerabilities [Wassermann and Su; PLDI07]

• For each vulnerability:
  – generate SQL + program path
  – check path consistency (Simplify)
  – solve string constraints (DPRLE)
Outline

• String Constraint Solving
• Preliminary Results

• Proposed Research
  – subset constraints
  – scalability through laziness
  – integer index operations
  – proof strategies
Scalability through Laziness

Idea: Cast constraint solving as a search problem. Traverse as little of the search space as possible.
Proposed Approach

datatype searchstate =
{ next : variable;
  states : variable→pos→status }
datatype status =
| Unknown of status
| StartsAt of nfastate→status
| Path of nfapath→status
Proposed Evaluation

• Within-domain performance comparison:
  – DPRLE
  – Hampi
  – CFG Analyzer
  – Rex

• Use previously-published benchmarks:
  – long strings task [Veanes et al.]
  – set difference task [Veanes et al.]
  – grammar intersection task [Kiezun et al.]
Outline

• String Constraint Solving
• Preliminary Results

• Proposed Research
  – subset constraints
  – scalability through laziness
  – integer index operations
  – proof strategies
Integer Index Operations

Idea:
Extend the lazy search-based approach to support integer index operations. Make use (if possible) of existing integer arithmetic models that support incremental solving.
Proposed Approach

- Explicitly-typed constraint language for strings and integer indices
- Support integer arithmetic on indices using an existing approach
Proposed Evaluation

• Compare to existing approach [Saxena et al.] where features overlap

• Develop PHP benchmark based on preliminary results

• Metrics: running time, proportion of testcases fully expressible
Outline

• String Constraint Solving
• Preliminary Results

• Proposed Research
  – subset constraints
  – scalability through laziness
  – integer index operations
  – proof strategies
Idea:
Develop a more general approach for formally verifying string decision procedures so that proof and algorithm can co-evolve.
Schedule

Automaton-Based [PLDI'09]
Bitvector-Based [ISSTA'09]
Lazy Search Algorithm
Lazy Search Journal
Search + Index Constraints
Proof Strategies (Optional)

2007 2008 2009 2010 2011 2012

Research Period
Publication Lag

today expected graduation
Conclusion

• Presented proposed research on decision procedures, focusing on:
  – expressive utility
  – scalability
  – correctness

• Research thrusts:
  – subset constraints
  – lazy search
  – integer index operations
  – proof strategies
We encourage difficult questions.
An Example

```c
void site_exec(char *cmd){
char *slash;
char *sp = (char*)strchr(cmd,' ');
/* sanitize the command-string */
while (sp &&
    (slash=strchr(cmd,'/')) &&
    (slash < sp))
    cmd = slash + 1;
}```
/* sanitize the command-string */
while (sp &&
    (slash=strchr(cmd,'/')) &&
    (slash < sp))
    cmd = slash + 1;

/bin/ls -ls

sp

cmd
void site_exec(char *cmd){
    char *slash;
    char *sp = (char*)strchr(cmd,' ');
    /* sanitize the command-string */
    while (sp &&
            (slash=strchr(cmd,'/')) &&
            (slash < sp))
        cmd = slash + 1;
    /bin/ls -ls
}
/* sanitize the command-string */
while (sp &&

(slash=strchr(cmd,'/')) &&
(slash < sp))

cmd = slash + 1;

/bin/ls  -ls\0

/cmd

/sp

/slash
/* sanitize the command-string */
while (sp &&
        (slash=strchr(cmd,'/')) &&
        (slash < sp))
    cmd = slash + 1;

/bin/ls -ls

\0

cmd
slash
sp
void site_exec(char *cmd) {
    char *slash = NULL;
    char *sp = (char*) strchr(cmd, ' ');
    /* sanitize the command-string */
    while (sp &&
           (slash=strchr(cmd, '/')) &&
           (slash < sp)) {
        cmd = slash + 1;
        sp = strchr(cmd, ' ');
    }
}
```c
void site_exec(char *cmd) {
    char *slash;
    char *sp = (char*) strchr(cmd, ' ');
    /* sanitize the command-string */
    while (sp &&
        (slash=strchr(cmd, '/')) &&
        (slash < sp))
        cmd = slash + 1;
    
    slash=0
}
```
void site_exec(char *cmd) {
    char *slash;
    char *sp = (char*) strchr(cmd, ' ');
    /* sanitize the command-string */
    while (sp &&
        (slash=strchr(cmd, '/') &&
         (slash < sp))
        cmd = slash + 1;

    string c ∈ Σ*
    index sp := findfirst(cmd, ' ');
    string c2 := cmd[:sp]
    index slsh := findlast(cmd2, '/ ')
    string c3 := cmd[slash + 1:]
Example: Some Queries

Can *cmd* contain '//'?

Can the substring between *cmd* and *sp* contain '/bin/rm'?