Verifying String Manipulation Programs

Pierre Flener

ASTRA Research Group
on Combinatorial Optimisation
Uppsala University
Sweden

CP-AI-OR 2015:
Master Class on Verification,
18 and 19 May 2015

Outline

Outline

String Constraint Problems

Constraints on strings occur in a wide variety of real-world application areas, such as:

- Web security: SQL code injection & XSS vulnerabilities
- Program testing: program test-case generation
- Program analysis
- Model checking
- Database testing
- Interactive configuration: of bikes or radar systems, say
- Biology: stem loops & pseudo-knots in bio-sequences
- Image processing: scene analysis
- Natural language processing: morphological analysis
- ...
- Crossword puzzle design

It is important to solve string problems efficiently & reliably.

Outline

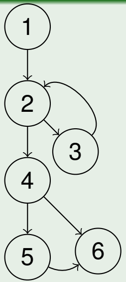
Program Testing

White-box test-case generation:

1. Infer the control flow graph from the code P under test.
2. Generate a path constraint,
   for every path in the control flow graph,
   under some code coverage criterion.
3. Infer a test case from a solution to each satisfiable path constraint, that is for each feasible path.
4. Use the tests for regression testing when P evolves.

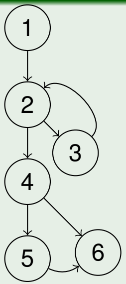
Example (Program Test-Case Generation)

```
function doSomething(s) {
  n = s.len();
  while (n mod 2 != 0)
    { t += s[n-1]; n = n/2; }
  if (s.match(/^(a*b)d\1$/))
    t = s.substr(1,s.len()/2);
  return t;
}
```



Example (Program Test-Case Generation)

```
0  function doSomething(s) {
1    n = s.len();
2    while (n mod 2 != 0)
3      { t += s[n-1]; n = n/2; }
4    if (s.match(/^(a*b)d\1$/))
5      t = s.substr(1,s.len()/2);
6    return t;
7  }
```

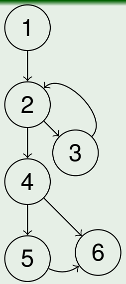


For the path $1 \rightarrow 2 \rightarrow 4 \rightarrow 5 \rightarrow 6$, a corresponding path constraint is as follows, where $\mathbb{S}$ is the type of strings:

$$\exists s, t \in \mathbb{S} : \exists n \in \mathbb{N} : n = |s| \wedge n \bmod 2 = 0$$
$$\wedge\ s \in \text{Language}((a^{*}b)d\backslash 1) \wedge t = s[1:n/2]$$

Example (Program Test-Case Generation)

```
0  function doSomething(s) {
1    n = s.len();
2    while (n mod 2 != 0)
3      { t += s[n-1]; n = n/2; }
4    if (s.match(/^(a*b)d\1$/))
5      t = s.substr(1,s.len()/2);
6    return t;
7  }
```



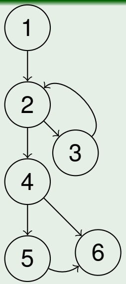
For the path $1 \to 2 \to 4 \to 5 \to 6$, a corresponding path constraint is as follows, where $\mathbb{S}$ is the type of strings:

$$\exists s, t \in \mathbb{S} : \exists n \in \mathbb{N} : n = |s| \wedge n \bmod 2 = 0$$
$$\wedge\ s \in \mathrm{Language}((a^*b)d\backslash 1) \wedge t = s[1:n/2]$$

This is unsatisfiable: no test case is needed for this path.

Example (Program Test-Case Generation)

```
0  function doSomething(s) {
1    n = s.len();
2    while (n mod 2 != 0)
3      { t += s[n-1]; n = n/2; }
4    if (s.match(/^(a*b)d\1$/))
5      t = s.substr(1,s.len()/2);
6    return t;
7  }
```



For the path $1 \to 2 \to 4 \to 5 \to 6$, a corresponding path constraint is as follows, where $\mathbb{S}$ is the type of strings:

$$\exists s, t \in \mathbb{S} : \exists n \in \mathbb{N} : n = |s| \land n \bmod 2 = 0$$
$$\land\ s \in \mathrm{Language}((\mathrm{a}^*\mathrm{b})\mathrm{d}\backslash 1) \land t = s[1 : n/2]$$

This is unsatisfiable: no test case is needed for this path. The path constraint for $1 \to 2 \to 4 \to 6$ *is* satisfiable, say with $\texttt{s} = $ "ab" as input, helping reveal that $\texttt{t}$ is uninitialised.

Outline

UPPSALA UNIVERSITET

Web Security

Web applications provide services in retail, banking, etc, in two-way communication with the (potential) customer:

- Read product catalogue, look up account balance, . . .
- Write review, add to cart, initiate wire transfer, . . .
- Authenticate identity, enter credit card information, . . .

Web Security

Web applications provide services in retail, banking, etc, in two-way communication with the (potential) customer:

- Read product catalogue, look up account balance, . . .
- Write review, add to cart, initiate wire transfer, . . .
- Authenticate identity, enter credit card information, . . .

Security and privacy threats arise:

- Theft of personal information, such as login credentials
- Gain of control over server
- Erasure of database
- . . .

This may lead to large financial loss or prestige loss.

UPPSALA
UNIVERSITET

Main Cause of Web Vulnerabilities

A weak spot in web applications is the processing of user inputs, especially when submitted as free-form strings!

Strings form a crucial datatype of scripting languages:

- JavaScript
- PHP
- ...

Hence there is a lot of focus on the analysis, testing, and verification of string manipulation programs.

Note that firewalls and cryptography are orthogonal forms of defence, because the flaws lie in the scripts themselves!

Example (Web Security)

Consider a message board application, with HTML form:

```
User identifier: "..."
Password: "..."
...
1. Display the messages with topic "..."
2. Post the message "..." with topic "..."
```

The user input is sent to a server via a URL, such as

http://bbs.org/?uid=laia&form=disp&top=cpaior

or

http://bbs.org/?uid=laia&form=post&msg=hola&top=cpaior

for retrieval from, or storage in, a database with the board.

Example (SQL Code Injection Vulnerability – part 1)

Consider the following snippet of a server-side PHP script:

```
$top = $_GET['top'];
$sel = "SELECT * FROM board WHERE topic='$top'";
$dat = mysql_query($sel);
```

The purpose is to retrieve the messages of a given topic.

Example (SQL Code Injection Vulnerability – part 1)

Consider the following snippet of a server-side PHP script:

```
$top = $_GET['top'];
$sel = "SELECT * FROM board WHERE topic='$top'";
$dat = mysql_query($sel);
```

The purpose is to retrieve the messages of a given topic. Messages of all topics are shown if an attacker enters topic

`gotcha' OR 'b'='b`

Example (SQL Code Injection Vulnerability – part 1)

Consider the following snippet of a server-side PHP script:

```
$top = $_GET['top'];
$sel = "SELECT * FROM board WHERE topic='$top'";
$dat = mysql_query($sel);
```

The purpose is to retrieve the messages of a given topic. Messages of all topics are shown if an attacker enters topic

`gotcha' OR 'b'='b`

Generate a constraint for this attack pattern:

$$\exists u, t \in \mathbb{S} : \exists f \in \{\text{disp}, \text{post}\} : |u| \geq 1 \wedge f = \text{disp}$$
$$\wedge\ t \in \text{Language}((\texttt{a-Z})^{+}\texttt{'␣OR␣'}((\texttt{a-Z})^{+})\texttt{'='}\backslash 1)$$

Example (SQL Code Injection Vulnerability – part 1)

Consider the following snippet of a server-side PHP script:

```
$top = $_GET['top'];
$sel = "SELECT * FROM board WHERE topic='$top'";
$dat = mysql_query($sel);
```

The purpose is to retrieve the messages of a given topic. Messages of all topics are shown if an attacker enters topic

`gotcha' OR 'b'='b`

Generate a constraint for this attack pattern:

$$\exists u, t \in \mathbb{S} : \exists f \in \{\text{disp}, \text{post}\} : |u| \geq 1 \land f = \text{disp}$$
$$\land\ t \in \text{Language}((\texttt{a-Z})^{+}\texttt{'␣OR␣'}((\texttt{a-Z})^{+})\texttt{'='}\backslash 1)$$

Its satisfaction helps us construct an attack vector, say:

`[uid="u", form="disp", top="a' OR 'b'='b"]`

Example (SQL Code Injection Vulnerability – part 1)

Consider the following snippet of a server-side PHP script:

```
$top = $_GET['top'];
$sel = "SELECT * FROM board WHERE topic='$top'";
$dat = mysql_query($sel);
```

The purpose is to retrieve the messages of a given topic. Messages of all topics are shown if an attacker enters topic

`gotcha' OR 'b'='b`

Generate a constraint for this attack pattern:

$$\exists u, t \in \mathbb{S} : \exists f \in \{\text{disp}, \text{post}\} : |u| \geq 1 \land f = \text{disp}$$
$$\land\ t \in \text{Language}((\texttt{a-Z})^{+}\texttt{'␣OR␣'}((\texttt{a-Z})^{+})\texttt{'='}\backslash 1)$$

Its satisfaction helps us construct an attack vector, say:

`[uid="u", form="disp", top="a' OR 'b'='b"]`

Input strings must be sanitised before going to the DBMS.

Example (SQL Code Injection Vulnerability – part 1)

Consider the following snippet of a server-side PHP script:

```
$top = $_GET['top'];
$sel = "SELECT * FROM board WHERE topic='$top'";
$dat = mysql_query($sel);
```

The purpose is to retrieve the messages of a given topic.
Messages of all topics are shown if an attacker enters topic

`gotcha' OR 'b'='b`

Generate a constraint for this attack pattern:

$$\exists u, t \in \mathbb{S} : \exists f \in \{\text{disp}, \text{post}\} : |u| \geq 1 \wedge f = \text{disp}$$
$$\wedge\ t \in \text{Language}((\texttt{a-Z})^{+}\texttt{'␣OR␣'}((\texttt{a-Z})^{+})\texttt{'='}\backslash 1)$$

Its satisfaction helps us construct an attack vector, say:

`[uid="u", form="disp", top="a' OR 'b'='b"]`

Or, better, a prepared query should be made to the DBMS.

Example (SQL Code Injection Vulnerability – part 2)

Consider again the snippet of a server-side PHP script:

```
$top = $_GET['top'];
$sel = "SELECT * FROM board WHERE topic='$top'";
$dat = mysql_query($sel);
```

The purpose is to retrieve the messages of a given topic.

Example (SQL Code Injection Vulnerability – part 2)

Consider again the snippet of a server-side PHP script:

```
$top = $_GET['top'];
$sel = "SELECT * FROM board WHERE topic='$top'";
$dat = mysql_query($sel);
```

The purpose is to retrieve the messages of a given topic.
The message board is erased if an attacker enters the topic

```
gotcha'; DROP TABLE board; --
```

Example (SQL Code Injection Vulnerability – part 2)

Consider again the snippet of a server-side PHP script:

```
$top = $_GET['top'];
$sel = "SELECT * FROM board WHERE topic='$top'";
$dat = mysql_query($sel);
```

The purpose is to retrieve the messages of a given topic. The message board is erased if an attacker enters the topic

`gotcha'; DROP TABLE board; --`

Generate a constraint for this attack pattern:

$$\exists u, t \in \mathbb{S} : \exists f \in \{\text{disp}, \text{post}\} : |u| \geq 1 \wedge f = \text{disp}$$
$$\wedge\ t \in \text{Language}((\texttt{a-Z})^{+}\texttt{';\textvisiblespace DROP\textvisiblespace TABLE\textvisiblespace}(\texttt{a-Z})^{+}\texttt{;\textvisiblespace --})$$

Example (SQL Code Injection Vulnerability – part 2)

Consider again the snippet of a server-side PHP script:

```
$top = $_GET['top'];
$sel = "SELECT * FROM board WHERE topic='$top'";
$dat = mysql_query($sel);
```

The purpose is to retrieve the messages of a given topic.
The message board is erased if an attacker enters the topic

`gotcha'; DROP TABLE board; --`

Generate a constraint for this attack pattern:

$$\exists u, t \in \mathbb{S} : \exists f \in \{\text{disp}, \text{post}\} : |u| \geq 1 \land f = \text{disp}$$
$$\land\ t \in \text{Language}((\texttt{a-Z})^{+}\texttt{';␣DROP␣TABLE␣}(\texttt{a-Z})^{+}\texttt{;␣--})$$

Its satisfaction helps us construct an attack vector, say:

`[uid="u", form="disp", top="a'; DROP TABLE b; --"]`

Example (SQL Code Injection Vulnerability – part 2)

Consider again the snippet of a server-side PHP script:

```
$top = $_GET['top'];
$sel = "SELECT * FROM board WHERE topic='$top'";
$dat = mysql_query($sel);
```

The purpose is to retrieve the messages of a given topic. The message board is erased if an attacker enters the topic

`gotcha'; DROP TABLE board; --`

Generate a constraint for this attack pattern:

$$\exists u, t \in \mathbb{S} : \exists f \in \{\text{disp}, \text{post}\} : |u| \geq 1 \land f = \text{disp}$$
$$\land\ t \in \text{Language}((\texttt{a-Z})^{+}\texttt{';␣DROP␣TABLE␣}(\texttt{a-Z})^{+}\texttt{;␣--})$$

Its satisfaction helps us construct an attack vector, say:

`[uid="u", form="disp", top="a'; DROP TABLE b; --"]`

Sanitise the input strings or, better, make a prepared query.

Example (Cross-Site Scripting Vulnerability, XSS-1)

Consider the following snippet of a server-side PHP script:

```
$user = $_GET['uid'];
echo "Dear $user, your message has been posted!";
```

The purpose is to display a message-posting confirmation.

Example (Cross-Site Scripting Vulnerability, XSS-1)

Consider the following snippet of a server-side PHP script:

```
$user = $_GET['uid'];
echo "Dear $user, your message has been posted!";
```

The purpose is to display a message-posting confirmation.

- A box appears in the victim's browser if lured by an attacker into clicking on a link that connects to the message board server with the following user identifier

```
Satan<script>alert("Gotcha!")</script>
```

Example (Cross-Site Scripting Vulnerability, XSS-1)

Consider the following snippet of a server-side PHP script:

```
$user = $_GET['uid'];
echo "Dear $user, your message has been posted!";
```

The purpose is to display a message-posting confirmation.

- A box appears in the victim's browser if lured by an attacker into clicking on a link that connects to the message board server with the following user identifier

  ```
  Satan<script>alert("Gotcha!")</script>
  ```

- But the attacker can execute any script in the victim's browser, with access to session data, permissions, cookies, browser credentials, etc.

Example (Cross-Site Scripting Vulnerability, XSS-1)

Consider the following snippet of a server-side PHP script:

```
$user = $_GET['uid'];
echo "Dear $user, your message has been posted!";
```

The purpose is to display a message-posting confirmation.

- A box appears in the victim's browser if lured by an attacker into clicking on a link that connects to the message board server with the following user identifier

  ```
  Satan<script>alert("Gotcha!")</script>
  ```

- But the attacker can execute any script in the victim's browser, with access to session data, permissions, cookies, browser credentials, etc.

Input strings must be sanitised before generating HTML.

Example (Cross-Site Scripting Vulnerability, XSS-2)

Consider the following snippets of a server-side PHP script:

```
$msg = $_GET['msg']; $top = $_GET['top'];
$ins = "INSERT INTO board VALUES('$msg','$top')";
$res = mysql_query($ins);

$sel = "SELECT * FROM board WHERE topic='$top'";
$dat = mysql_query($sel);
while($r=mysql_fetch_assoc($dat)){echo $r['msg'];}
```

The purposes are to post a message & display messages.

Example (Cross-Site Scripting Vulnerability, XSS-2)

Consider the following snippets of a server-side PHP script:

```
$msg = $_GET['msg']; $top = $_GET['top'];
$ins = "INSERT INTO board VALUES('$msg','$top')";
$res = mysql_query($ins);

$sel = "SELECT * FROM board WHERE topic='$top'";
$dat = mysql_query($sel);
while($r=mysql_fetch_assoc($dat)){echo $r['msg'];}
```

The purposes are to post a message & display messages.

1. Assume an attacker posts the topic-“`CPAIOR`” message

   `Gotcha!<script>alert("Gotcha!")</script>`

Example (Cross-Site Scripting Vulnerability, XSS-2)

Consider the following snippets of a server-side PHP script:

```
$msg = $_GET['msg']; $top = $_GET['top'];
$ins = "INSERT INTO board VALUES('$msg','$top')";
$res = mysql_query($ins);

$sel = "SELECT * FROM board WHERE topic='$top'";
$dat = mysql_query($sel);
while($r=mysql_fetch_assoc($dat)){echo $r['msg'];}
```

The purposes are to post a message & display messages.

1. Assume an attacker posts the topic-"`CPAIOR`" message

   `Gotcha!<script>alert("Gotcha!")</script>`

2. A box appears in any victim's browser if innocently retrieving all the messages with the topic "`CPAIOR`".

Example (Cross-Site Scripting Vulnerability, XSS-2)

Consider the following snippets of a server-side PHP script:

```
$msg = $_GET['msg']; $top = $_GET['top'];
$ins = "INSERT INTO board VALUES('$msg','$top')";
$res = mysql_query($ins);

$sel = "SELECT * FROM board WHERE topic='$top'";
$dat = mysql_query($sel);
while($r=mysql_fetch_assoc($dat)){echo $r['msg'];}
```

The purposes are to post a message & display messages.

1. Assume an attacker posts the topic-"`CPAIOR`" message

   `Gotcha!<script>alert("Gotcha!")</script>`

2. A box appears in any victim's browser if innocently retrieving all the messages with the topic "`CPAIOR`".

Input strings must be sanitised before writing SQL or HTML.

Outline

Outline

Decision Variables

Consider a string type $\mathbb{S}$, for strings over a character type $\mathbb{C}$.

Kinds of string variables:

- Fixed length ℓ: we write $\mathbb{S} = \mathbb{C}^\ell$
- Bounded length ℓ: we write $\mathbb{S} = \mathbb{C}^{\leq \ell}$
- Unbounded but finite length: we write $\mathbb{S} = \mathbb{C}^*$

The domain of a string variable is a set of strings, that is a language over an alphabet Σ that is a finite subset of $\mathbb{C}$.

If $\mathbb{S} = \mathbb{B}^\ell$ for some ℓ, then we speak of bit-vector variables.

We can also imagine a type of regular-expression variables.

UPPSALA UNIVERSITET

Outline

Constraints

Consider decision variables $s_j \in \mathbb{S}$, and $c_j \in \mathbb{C}$, and $i_j \in \mathbb{N}$:

- $\textsc{Equal}(s_1, s_2)$ if s_1 and s_2 are equal, that is $s_2 = s_1$
- $\textsc{Reverse}(s_1, s_2)$ if $s_1 = c_1 c_2 \cdots c_n$ and $s_2 = c_n \cdots c_2 c_1$
- $\textsc{Concat}(s_1, s_2, s)$ if $s_1 \oplus s_2 = s$, with concatenation $\oplus$
- $\textsc{SubString}(s_1, i_1, i_2, s)$ if $s_1[i_1 : i_2] = s$
- $\textsc{CharacterAt}(s, i, c)$ if $\textsc{SubString}(s, i, i, \text{“}c\text{”})$
- $\textsc{Length}(s, i)$ if s has i characters, that is $|s| = i$
- $\textsc{Regular}(s, \mathcal{R})$ if s is a word of a regular language $\mathcal{R}$, given by a regular expression or a finite automaton
- $\textsc{ContextFree}(s, \mathcal{F})$ if s is a word of a context-free language $\mathcal{F}$, given by a context-free grammar
- $[\textsc{Regular}]\textsc{ReplaceAll}(s_1, s_2, s_3, s)$ if $s_1[s_2 \rightsquigarrow s_3] = s$
- $\textsc{Count}(s, [c_1, \ldots, c_n], [i_1, \ldots, i_n])$ if in s all c_j occur i_j times
- . . . , especially for bit-vector variables

Constraints

Consider decision variables $s_j \in \mathbb{S}$, and $c_j \in \mathbb{C}$, and $i_j \in \mathbb{N}$:

- $\textsc{Equal}(s_1, s_2)$ if s_1 and s_2 are equal, that is $s_2 = s_1$
- $\textsc{Reverse}(s_1, s_2)$ if $s_1 = c_1 c_2 \cdots c_n$ and $s_2 = c_n \cdots c_2 c_1$
- $\textsc{Concat}(s_1, s_2, s)$ if $s_1 \oplus s_2 = s$, with concatenation $\oplus$
- $\textsc{SubString}(s_1, i_1, i_2, s)$ if $s_1[i_1 : i_2] = s$
- $\textsc{CharacterAt}(s, i, c)$ if $\textsc{SubString}(s, i, i, \text{“}c\text{”})$
- $\textsc{Length}(s, i)$ if s has i characters, that is $|s| = i$
- $\textsc{Regular}(s, \mathcal{R})$ if s is a word of a regular language $\mathcal{R}$, given by a regular expression or a finite automaton
- $\textsc{ContextFree}(s, \mathcal{F})$ if s is a word of a context-free language $\mathcal{F}$, given by a context-free grammar
- $[\textsc{Regular}]\textsc{ReplaceAll}(s_1, s_2, s_3, s)$ if $s_1[s_2 \rightsquigarrow s_3] = s$
- $\textsc{Count}(s, [c_1, ..., c_n], [i_1, ..., i_n])$ if in s all c_j occur i_j times
- ..., especially for bit-vector variables

Example

The path constraint of the test-case generation example:

$$\exists s,t \in \mathbb{S} : \exists n \in \mathbb{N} : n = |s| \wedge n \bmod 2 = 0$$
$$\wedge\ s \in \text{Language}((\text{a}^*\text{b})\text{d}\backslash 1) \wedge t = s[1:n/2]$$

can now be relationally modelled as follows:

$$\exists s, s_1, s_2, t \in \mathbb{S} : \exists n, m \in \mathbb{N} : \textsc{Length}(s,n) \wedge \textsc{Mod}(n,2,0)$$
$$\wedge\ \textsc{Regular}(s_1, \text{Language}(\text{a}^*\text{b}))$$
$$\wedge\ \textsc{Concat}(\text{“d”}, s_1, s_2) \wedge \textsc{Concat}(s_1, s_2, s)$$
$$\wedge\ \textsc{Div}(n,2,m) \wedge \textsc{SubString}(s,1,m,t)$$

The back-reference $\backslash 1$ was modelled away using CONCAT.
Back-references may yield context-sensitive languages.

Since many string constraints are total-function constraints, it seems best to model with functions and relations, and to translate into relational form only for solving.

Constraints on Unbounded-Length Strings

Definition

A word equation is of the form $\ell = r$, where ℓ and r are concatenations of string constants and string variables of unbounded but finite length.

Example

The word equation $\exists s, t \in \mathbb{S} : s \oplus \text{“ab”} \oplus t = t \oplus \text{“ba”} \oplus s$

Constraints on Unbounded-Length Strings

Definition

A word equation is of the form $\ell = r$, where ℓ and r are concatenations of string constants and string variables of unbounded but finite length.

Example

The word equation $\exists s, t \in \mathbb{S} : s \oplus \text{“ab”} \oplus t = t \oplus \text{“ba”} \oplus s$ has $\{s \mapsto \text{“a”},\ t \mapsto \text{“aa”}\}$ as non-unique solution.

Constraints on Unbounded-Length Strings

Definition

A word equation is of the form $\ell = r$, where ℓ and r are concatenations of string constants and string variables of unbounded but finite length.

Example

The word equation $\exists s, t \in \mathbb{S} : s \oplus \text{“ab”} \oplus t = t \oplus \text{“ba”} \oplus s$ has $\{s \mapsto \text{“a”},\ t \mapsto \text{“aa”}\}$ as non-unique solution.

Satisfiability of (systems of) word equations is decidable, but only exponential-time algorithms exist [Makanin, 1977].

In the presence of even small sets of other constraints, decidability is lost or open.

Outline

Outline

Prehistory

Custom string solvers:

- Based on some constraint solving technology: no, ad hoc
- Usage: one tool, for a targeted application area
- Modelling language: none
  - Separation of constraint generation and solving: no
  - Separation of constraints and instance data: no
- Variable types: one kind of strings, encoded integers
- Collection of string constraints: limited to targeted area
- Efficiency: slow
- Reliability: no correctness guarantees
- Documentation: not needed
- Maintenance: unknown

Middle Ages

Extensions, say *Hampi* and *Kaluza*, **of existing solvers:**

- Based on some constraint solving technology: yes, with encoding of string variables using native types
- Usage: hundreds of tools, only in the verification area
- Modelling language: yes, but low-level, solver-specific
  - Separation of constraint generation and solving: yes
  - Separation of constraints and instance data: no
- Variable types: one kind of strings, native integers
- Collection of string constraints: limited to verification
- Efficiency: orders of magnitude more efficient, but ...
- Reliability: theoretically yes, but in practice not
- Documentation: poor
- Maintenance: rare

Renaissance (today!)

Extensions, say *CVC4* and *Z3-Str2*, **of existing solvers:**

- Based on some constraint solving technology: yes, with native representation of string variables
- Usage: hundreds of tools (?), only in verification area
- Modelling language: yes, but low-level, techno-specific
  - Separation of constraint generation and solving: yes
  - Separation of constraints and instance data: no
- Variable types: some combinations
- Collection of string constraints: limited to verification
- Efficiency: even more efficient
- Reliability: yes
- Documentation: better
- Maintenance: frequent

UPPSALA UNIVERSITET

Limitations of some String Solvers

- Emmi *et al.* @ ISSTA 2007:
  "Our constraint solver is approximate, in that it assumes that arithmetic and string constraints do not interact, and therefore may fail to find satisfying assignments"

UPPSALA UNIVERSITET

Limitations of some String Solvers

- Emmi *et al.* @ ISSTA 2007:
  "Our constraint solver is approximate, in that it assumes that arithmetic and string constraints do not interact, and therefore may fail to find satisfying assignments"

- Paper on *Hampi* @ ISSTA 2009:
  "A *Hampi* input must declare a *single* string variable"

  Paper on *Kaluza* @ SP 2010:
  "over multiple variable-length string inputs"

Outline

Fixed-Length String Variables in CP

Many additional constraints:

- MDD$(s, \mathcal{M})$ if s is a word of a regular language $\mathcal{M}$, given by a multi-valued decision diagram
- TABLE$(s, \mathcal{T})$ if s is a word of a regular language $\mathcal{T}$, given by a table
- AUTOMATON$(s, \mathcal{S})$ if s is a word of a context-sensitive language $\mathcal{S}$, given by an automaton with accumulators
- ...

Fixed-Length String Variables in CP

Many additional constraints:

- $\text{MDD}(s, \mathcal{M})$ if s is a word of a regular language $\mathcal{M}$, given by a multi-valued decision diagram
- $\text{TABLE}(s, \mathcal{T})$ if s is a word of a regular language $\mathcal{T}$, given by a table
- $\text{AUTOMATON}(s, \mathcal{S})$ if s is a word of a context-sensitive language $\mathcal{S}$, given by an automaton with accumulators
- ...

Approaches:

- Use fixed-length array of variables over alphabet $\Sigma \subseteq \mathbb{C}$
- Michel & Van Hentenryck @ CP 2012: Bit-vector variables, with a native implementation
- Quimper & Walsh @ CSCLP 2005, ..., Monette *et al.* @ AAAI 2014: Tuple variables, implementable by tuple lists, intervals, mappings, polyhedra, bounded-width MDDs, etc.

Bounded-Length String Variables in CP

- Maher @ CP-AI-OR 2009: Bounded open constraints: The solver adds variables at the end when needed.
- Padding representation, e.g., He *et al.* @ CP 2013:
  - Use a fixed-length string variable, whose length is the given upper bound
  - Add a dummy character $\perp$ to the alphabet Σ
  - Re-constrain occurrences of $\perp$ towards end of a string:
    - Replace $\textsc{Regular}(s, \mathcal{R})$ by $\textsc{Regular}(s, \mathcal{R}\perp^*)$
    - Change grammars of ContextFree constraints
    - Decompose Equal, Concat, etc.
- Scott *et al.* @ CP-AI-OR 2015: Bounded representation
  ☞ Wednesday at 11:25

Unbounded-Length String Variables in CP

- Golden & Pang @ CP 2003: Automaton representation:
  - The domain of a string variable is the set of all and only its values in the solutions to the system of constraints posted so far; it is represented by a finite automaton, initially the one for Σ^*
  - Exponential-space automata
  - Exponential-time propagators with automata operations
  - No search!

Outline

String Solving and SMT Technology

The Nelson-Oppen combination framework, used by most SMT solvers, combines decision procedures for theories T_1 and T_2 into a decision procedure for $T_1 \uplus T_2$ using only term equality propagation between T_1 and T_2, where T_1 and T_2 can only share variables as well as the $=$ and $\neq$ predicates, and where satisfiable formulae must have stably infinite satisfying valuations (called *models*).

☞ Stable infinity limits SMT to unbounded-length string variables, whereas CP can also have fixed/bounded-length string variables and finite-domain integer variables.

UPPSALA UNIVERSITET

String Solving and SMT Technology

The Nelson-Oppen combination framework, used by most SMT solvers, combines decision procedures for theories T_1 and T_2 into a decision procedure for $T_1 \uplus T_2$ using only term equality propagation between T_1 and T_2, where T_1 and T_2 can only share variables as well as the $=$ and $\neq$ predicates, and where satisfiable formulae must have stably infinite satisfying valuations (called *models*).

☞ Stable infinity limits SMT to unbounded-length string variables, whereas CP can also have fixed/bounded-length string variables and finite-domain integer variables.

- Theorem provers are geared at problems where proven unsatisfiability is hoped for, such as unreachability.
- CP solvers are geared at problems where satisfiability is acceptable, such as test case generation.

So CP and SMT solvers have complementary strengths!

Also:

- For strings of unbounded but finite length, decidability is lost or open for many collections of predicates.

  ☞ For an SMT solver, incompleteness is a problem, unlike for a CP solver.

- SMT-Lib format: proposals for string variables, such as "An SMT-LIB Format for Sequences and Regular Expressions" and those of *CVC4* and *Z3-Str2*, exist, but they do not include CONTEXTFREE and COUNT.

Outline

Outline

Scott *et al*. @ CP-AI-OR 2015:

☞ Wednesday at 11:25

UPPSALA UNIVERSITET

Outline

1. Motivation
   - Program Testing
   - Web Security
2. Definitions
   - Decision Variables
   - Constraints
3. History
   - Eras
   - CP Technology
   - SMT Technology
4. **First Results**
   - Bounded Representation
   - **Experiments**
5. Outlook

Argument for Higher-Level Modelling

Sushi @ Formal Aspects of Computing 2013

Sushi solves word equations on unbounded-length strings.

Simple linear string equation (SLSE) $\ell \equiv r$:

- ℓ has exactly one occurrence of each string variable
- ℓ has CONCAT, REGULARREPLACEALL, SUBSTRING
- r is a regular expression: REGULAR(s, r) if ℓ fixes s
- functional notation

Typically, reg. exp. r is an attack pattern of code injection.

Example: SLSE 1: $\exists s \in \mathbb{S} : s \oplus \mathrm{a}^n \equiv (\mathrm{a}|\mathrm{b})^{2n}$, for $n \in \mathbb{N}$

Argument for Higher-Level Modelling

Sushi @ Formal Aspects of Computing 2013

Sushi solves word equations on unbounded-length strings.

Simple linear string equation (SLSE) $\ell \equiv r$:

- ℓ has exactly one occurrence of each string variable
- ℓ has CONCAT, REGULARREPLACEALL, SUBSTRING
- r is a regular expression: $\textsc{Regular}(s, r)$ if ℓ fixes s
- functional notation

Typically, reg. exp. r is an attack pattern of code injection.

Example: SLSE 1: $\exists s \in \mathbb{S} : s \oplus \mathrm{a}^n \equiv (\mathrm{a}|\mathrm{b})^{2n}$, for $n \in \mathbb{N}$

If modelled at a higher level of abstraction and relationally, then the SLSE problem structure gives a model whose constraint network is Berge-acyclic, hence domain consistency on each constraint gives backtrack-free solving!

Sushi Benchmark

Sushi is automaton-based: all solutions are found together.

For $n \leq 37$, *Sushi* beats the bounded-length solver *Kaluza*.

Runtimes in seconds to first solutions (fastest in **bold**):

	$n = 37$			$n = 50$			$n = 100$		
	bounded	pad	*Sushi*	bounded	pad	*Sushi*	bounded	pad	*Sushi*
Eq. 1	**0.02**	0.05	0.30	**0.02**	0.07	1.11	**0.09**	0.24	2.56
Eq. 2	**<0.01**	**<0.01**	0.37	**<0.01**	**<0.01**	0.88	**0.01**	0.02	19.24
Eq. 3	**0.01**	0.03	0.29	**0.01**	0.03	0.64	**0.02**	0.09	1.14
Eq. 4	**<0.01**	0.01	42.16	**<0.01**	0.03	>300	**0.06**	0.07	>300
Eq. 5	**<0.01**	**<0.01**	1.56	**<0.01**	**<0.01**	2.93	**<0.01**	0.02	6.37

All instances run without backtracks in the bounded-length and padding CP approaches, which have the same search trees upon the same deterministic search, under *Gecode*.

Kaluza Benchmark

Instances were generated from real-world web applications.

Runtimes in seconds and backtracks (best in **bold**):

	bounded		padding		*Kaluza*
instance name	runtime	backtracks	runtime	backtracks	runtime
`concat`	**0.003**	**0**	0.008	**0**	0.088
`indexof`	**0.003**	**0**	0.010	**0**	1.560
`bettermatch1`	**0.002**	**0**	0.005	**0**	0.223
`bettermatch2`	**0.003**	**0**	**0.003**	**0**	0.192
`streq`	**0.003**	**0**	0.006	**0**	0.077
`replace`	**0.006**	**30**	0.019	**30**	0.364
50,000+ instances	**<0.010**	**<50**	<0.020	**<50**	<0.500

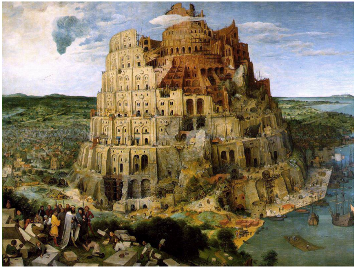
Z3-Str was only applied to REGULAR-less instance versions.

Attitude on Benchmarks

Almost no runtimes above one second are reported:

- Are most problems closed?
- Are there no tougher instances?
- Is real-time solvability only what they can solve fast?

Before the use of modelling languages, benchmark instances were often shared but are hard to translate.



UPPSALA UNIVERSITET

Outline

Future

Extensions of existing solvers:

- Based on constraint solving technology: yes, with native representation of string variables
- Usage: hundreds of tools, in all application areas
- Modelling language: higher-level, techno-independent
  - Separation of constraint generation and solving: yes
  - Separation of constraints and instance data: yes
- Variable types: any combination
- Collection of string constraints: comprehensive
- Efficiency: yet more efficient
- Reliability: yes
- Documentation: good
- Maintenance: frequent

CP seen by Analysis, Testing, & Verification

Paper on *Hampi* @ ISSTA 2009:

> "A variety of problems involve string constraints, and there is an extensive literature on the theoretical study of these problems [citations to REGULAR @ CP04 and CONTEXTFREE @ CP06]. Our work is focused on efficient techniques for a practical string-constraint solver that is usable as a library and is sufficiently expressible (*sic*) to support a large variety of applications."

CP seen by Analysis, Testing, & Verification

Paper on *Hampi* @ ISSTA 2009:

> "A variety of problems involve string constraints, and there is an extensive literature on the theoretical study of these problems [citations to REGULAR @ CP04 and CONTEXTFREE @ CP06]. Our work is focused on efficient techniques for a practical string-constraint solver that is usable as a library and is sufficiently expressible (*sic*) to support a large variety of applications."

☞ CP technology needs to help modellers package up a model front-ended by a domain-specific language!

UPPSALA UNIVERSITET

Acknowledgements

Many thanks to:

- Jun He
- Jean-Noël Monette
- Justin Pearson
- Philipp Rümmer
- Christian Schulte
- Joseph D. Scott
- the Swedish Research Council (VR): grant 2011-6133

More questions?