Automata-based Model Counting
Model Counting String Constraint Solver

**INPUT**

string constraint: \( C \)

**OUTPUT**

counting function: \( f_c \)

length bound: \( k \)

\# of strings with length \( \leq k \) for which \( C \) evaluates to true

Aydin et al., Automata-based Model Counting for String Constraints. (CAV’15)
Automata Based Counter (ABC)
A Model Counting String Constraint Solver

INPUT
string constraint: \( C \)

Automata-Based model Counting string constraint solver (ABC)

OUTPUT
counting function: \( f_c \)
length bound: \( k \)
\# of strings with length \( \leq k \) for which \( C \) evaluates to true

Aydin et al., Automata-based Model Counting for String Constraints. (CAV’15)
String Constraint Language

\[ C \rightarrow bterm \]

\[ bterm \rightarrow v \mid true \mid false \]
\[ \quad \mid \neg bterm \mid bterm \land bterm \mid bterm \lor bterm \mid (bterm) \]
\[ \quad \mid sterm = sterm \]
\[ \quad \mid \text{match}(sterm, sterm) \]
\[ \quad \mid \text{contains}(sterm, sterm) \]
\[ \quad \mid \text{begins}(sterm, sterm) \]
\[ \quad \mid \text{ends}(sterm, sterm) \]
\[ \quad \mid iterm = iterm \mid iterm < iterm \mid iterm > iterm \]

\[ iterm \rightarrow v \mid n \]
\[ \quad \mid iterm + iterm \mid iterm - iterm \mid iterm \times n \mid (iterm) \]
\[ \quad \mid \text{length}(sterm) \mid \text{toint}(sterm) \]
\[ \quad \mid \text{indexOf}(sterm, sterm) \]
\[ \quad \mid \text{lastIndexOf}(sterm, sterm) \]

\[ sterm \rightarrow v \mid \varepsilon \mid s \]
\[ \quad \mid sterm.sterm \mid sterm|sterm \mid sterm^* \mid (sterm) \]
\[ \quad \mid \text{charAt}(sterm, iterm) \mid \text{toString}(iterm) \]
\[ \quad \mid \text{toUpper}(sterm) \mid \text{toLower}(sterm) \]
\[ \quad \mid \text{substring}(sterm, iterm, iterm) \]
\[ \quad \mid \text{replaceFirst}(sterm, sterm, sterm) \]
\[ \quad \mid \text{replaceLast}(sterm, sterm, sterm) \]
\[ \quad \mid \text{replaceAll}(sterm, sterm, sterm) \]
ABC: Constraint language

- A more compact notation

\[
\begin{align*}
\varphi & \rightarrow \varphi \land \varphi \mid \varphi \lor \varphi \mid \neg \varphi \mid \varphi_0 \mid \varphi_s \mid T \mid \bot \\
\varphi_0 & \rightarrow \beta = \beta \mid \beta < \beta \mid \beta > \beta \\
\varphi_s & \rightarrow \gamma = \gamma \mid \gamma < \gamma \mid \gamma > \gamma \mid \text{match}(\gamma, \rho) \mid \text{contains}(\gamma, \gamma) \mid \text{begins}(\gamma, \gamma) \mid \text{ends}(\gamma, \gamma) \\
\beta & \rightarrow v_i \mid n \mid \beta + \beta \mid \beta - \beta \mid \beta \times n \\
& \quad \mid \text{length}(\gamma) \mid \text{toint}(\gamma) \mid \text{indexof}(\gamma, \gamma) \mid \text{lastindexof}(\gamma, \gamma) \\
\gamma & \rightarrow v_s \mid \rho \mid \gamma \cdot \gamma \mid \text{reverse}(\gamma) \mid \text{tostring}(\beta) \mid \text{charat}(\gamma, \beta) \mid \text{toupper}(\gamma) \mid \text{tolower}(\gamma) \\
& \quad \mid \text{substring}(\gamma, \beta, \beta) \mid \text{replacefirst}(\gamma, \gamma, \gamma) \mid \text{replacelast}(\gamma, \gamma, \gamma) \mid \text{replaceall}(\gamma, \gamma, \gamma) \\
\rho & \rightarrow \varepsilon \mid s \mid \rho \cdot \rho \mid \rho \cdot \rho \mid \rho^* 
\end{align*}
\]
# Example String Expressions

<table>
<thead>
<tr>
<th>String Expression</th>
<th>Constraint Language</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>s.length()</code></td>
<td><code>length(s)</code></td>
</tr>
<tr>
<td><code>s.isEmpty()</code></td>
<td><code>length(s) == 0</code></td>
</tr>
<tr>
<td><code>s.startsWith(t,n)</code></td>
<td>`0 \leq n \land n \leq</td>
</tr>
<tr>
<td><code>s.indexOf(t,n)</code></td>
<td>`indexOf(substring(s,n,</td>
</tr>
<tr>
<td><code>s.replaceAll(p,r)</code></td>
<td><code>replaceall(s,p,r)</code></td>
</tr>
<tr>
<td><code>strstrpos(s, t)</code></td>
<td><code>lastindexOf(s,t)</code></td>
</tr>
<tr>
<td><code>substr_replace(s, t, i, j)</code></td>
<td>`substring(s,0,i).t.substring(s,j,</td>
</tr>
<tr>
<td><code>strip_tags(s)</code></td>
<td>`replaceall(s, (&quot;&lt;a&gt;&quot;</td>
</tr>
<tr>
<td><code>mysql_real_escape_string</code></td>
<td><code>...replaceall(s,replaceall(s,&quot;\&quot;&quot;,&quot;\\\\&quot;)</code></td>
</tr>
<tr>
<td></td>
<td><code>,replaceall(s, &quot;\&quot;&quot;,&quot;\\\\&quot;)</code>,<code>&quot;&quot;&quot;, &quot;\&quot;&quot;)...</code></td>
</tr>
</tbody>
</table>
Model Counting String Constraint Solver

**INPUT**

string constraint: \( C \)

**OUTPUT**

counting function: \( f_c \)

length bound: \( k \)

\# of strings with length \( \leq k \) for which \( C \) evaluates to true

---

Aydin et al., Automata-based Model Counting for String Constraints. (CAV'15)
ABC in a nutshell

Automata-based constraint solving

Why?
ABC in a nutshell

Automata-based constraint solving

**Basic idea:**

Constructing an automaton for the set of solutions of a constraint reduces model counting problem to path counting!
Automata-based constraint solving

Generate automaton that accepts satisfying solutions for the constraint

ABC can handle both string and integer constraints

- Constraints over only string variables (e.g., v = “abcd”)
- Constraints over only integer variables (e.g., i = 2×j)
- Constraints over both string and integer variables (e.g., length(v) = i)
Automata-based constraint solving: expr, ¬

Basic string constraints are directly mapped to automata

\[ v = "ab" \]
\[ \text{match}(v, (ab)^*) \]
\[ \neg \text{match}(v, (ab)^*) \]

automata complement
Automata-based constraint solving: \( \text{expr}, \neg, \land, \lor \)

More complex constraints are solved by creating automata for subformulae then combining their results

\[ \neg \text{match}(v, (ab)^*) \land \text{length}(v) = 2 \]

automata product
Automata-based constraint solving: expr, \( \neg \), \( \land \), \( \lor \)

More complex constraints are solved by creating automata for subformulae then combining their results

\( \neg \text{match}(v, (ab)^*) \land \text{length}(v) = 2 \)
String Automata Construction: More Details

\[ C \equiv \neg (x \in (01)^*) \land LEN(x) = 2 \]
String Automata Construction

\[ C \equiv \neg(x \in (01)^*) \land LEN(x) = 2 \]
String Automata Construction

\[ C \equiv \neg(x \in (01)^*) \land LEN(x) = 2 \]
String Automata Construction

$$C \equiv \neg(x \in (01)^*) \land LEN(x) = 2$$
\[ C \equiv \neg(x \in (01)^*) \land \text{LEN}(x) = 2 \]
String Automata Construction

\[ C \equiv \neg(x \in (01)^*) \land LEN(x) = 2 \]
String Automata Construction

\[ C \equiv \neg (x \in (01)^*) \land \text{LEN}(x) = 2 \]
String Automata Construction

\[ C \equiv \neg(x \in (01)^*) \land LEN(x) = 2 \]
String Automata Construction

\[ C \equiv \neg (x \in (01)^*) \land \text{LEN}(x) = 2 \]
String Automata Construction

\[ C \equiv \neg(x \in (01)^*) \land LEN(x) = 2 \]
String Automata Construction

$C \equiv \neg(x \in (01)^*) \land LEN(x) = 2$
String Automata Construction

\[ C \equiv \neg(x \in (01)^*) \land \text{LEN}(x) = 2 \]
String Automata Construction

\[ C \equiv \neg(x \in (01)^*) \land LEN(x) = 2 \]
String Automata Construction

\[ C \equiv \neg(x \in (01)^*) \land LEN(x) = 2 \]
String Automata Construction

\[ C \equiv \neg(x \in (01)^*) \land LEN(x) = 2 \]
String Automata Construction

\[ C \equiv \neg(x \in (01)^*) \land LEN(x) = 2 \]
String Automata Construction

\[ C \equiv \neg (x \in (01)^*) \land LEN(x) = 2 \]
String Automata Construction

\[ C \equiv \neg(x \in (01)^*) \land LEN(x) = 2 \]
String Automata Construction

\[ C \equiv \neg (x \in (01)^*) \land LEN(x) = 2 \]
String Automata Construction

\[ C \equiv \neg(x \in (01)^*) \land LEN(x) = 2 \]
String Automata Construction

\[ C \equiv \neg (x \in (01)^*) \land \text{LEN}(x) = 2 \]
String Automata Construction

\[ C \equiv \neg(x \in (01)^*) \land LEN(x) = 2 \]

00, 10, 11
Relational constraints

- Relational constraints:
  - Constraints that involve multiple variables

- How do we handle relational constraints with automata?
Automata-based constraint solving: relational

For multi-variable constraints, generate an automaton for each variable

\[ v = t \quad v \neq t \quad v = t \land v \neq t \]
Automata-based constraint solving: relational

For multi-variable constraints, generate an automaton for each variable

\[ v = t \]
\[ v \neq t \]

\[ v = t \land v \neq t \]

Satisfiable!
Automata-based constraint solving: relational

Single track automata cannot precisely capture relational constraints

Generated automata significantly over-approximate # of satisfying solutions

Use multi-track automata
Multi-track automata

Multi-track automaton = DFA accepting tuples of strings

Each track represents the values of a single variable

Preserves relations among variables!
Multi-track automata

\( v = t \)

Padding symbol \( \lambda \not\in \Sigma \) used to align tracks of different length (appears at the end)

\( v \neq t \)

Correctly encodes the constraint

\( v = t \land v \neq t \)

Automata product
Relational String Constraints: Summary

- How to handle constraints with multiple string variables?
  - One approach is to use multiple single-track DFAs
    - One DFA per variable
  - Alternative approach: Use one multi-track DFAs
    - Each track represents the values of one string variable

- Using multi-track DFAs:
  - Identifies the relations among string variables
  - Improves the precision
  - Can be used to represent properties that depend on relations among string variables, e.g., $file = $usr.txt
Multi-track Automata

- Let $X$ (the first track), $Y$ (the second track), be two string variables
- $\lambda$ is the padding symbol
- A multi-track automaton that encodes the word equation:

$$Y = X.txt$$

(a,a), (b,b) …
Alignment

To conduct relational string analysis, we need to compute union or intersection of multi-track automata

- Intersection is closed under aligned multi-track automata
  - In an aligned multi-track automaton λs are right justified in all tracks, e.g., abλλ instead of aλbλ

However, there exist unaligned multi-track automata that are not equivalent to any aligned multi-track automata

- Use an alignment algorithm that constructs aligned automata which over or under approximates unaligned ones
  - Over approximation: Generates an aligned multi-track automaton that accepts a super set of the language recognized by the unaligned multi-track automaton
  - Under approximation: Generates an aligned multi-track automaton that accepts a subset of the language recognized by the unaligned multi-track automaton
Word Equations

- Word equations: Equality of two expressions that consist of concatenation of a set of variables and constants
  - Example: $X = Y . \text{txt}$

- Word equations and their combinations (using Boolean connectives) can be expressed using only equations of the form $X = Y . c, X = c . Y, c = X . Y, X = Y . Z$, Boolean connectives and existential quantification

- Construct multi-track automata from basic word equations
  - The automata should accept tuples of strings that satisfy the equation
  - Boolean connectives can be handled using intersection, union and complement
  - Existential quantification can be handled using projection
Word Equations to Automata

- Basic equations $X = Y \cdot c$, $X = c \cdot Y$, $c = X \cdot Y$ and their Boolean combinations can be represented precisely using multi-track automata.

- The size of the aligned multi-track automaton for $X = c \cdot Y$ is exponential in the length of $c$.

- The nonlinear equation $X = Y \cdot Z$ cannot be represented precisely using an aligned multi-track automaton.
Word Equations to Automata

- When we cannot represent an equation precisely, we can generate an over or under-approximation of it

  - Over-approximation: The automaton accepts all string tuples that satisfy the equation and possibly more

  - Under-approximation: The automaton accepts only the string tuples that satisfy the equation but possibly not all of them

- We can implement a function \( \text{CONSTRUCT}(\text{equation}, \text{sign}) \)
  - Which takes a word equation and a sign and creates a multi-track automata that over or under-approximation of the equation based on the input sign
Integer Constraints

\[ C \rightarrow bterm \]

\[ bterm \rightarrow v \mid \text{true} \mid \text{false} \]
\[ -bterm \mid bterm \land bterm \mid bterm \lor bterm \mid (bterm) \]
\[ sterm = sterm \]
\[ \text{match}(sterm, sterm) \]
\[ \text{contains}(sterm, sterm) \]
\[ \text{begins}(sterm, sterm) \]
\[ \text{ends}(sterm, sterm) \]
\[ iterm = iterm \mid iterm < iterm \mid iterm > iterm \]

\[ iterm \rightarrow v \mid n \]
\[ iterm + iterm \mid iterm - iterm \mid iterm \times n \mid (iterm) \]
\[ \text{length}(sterm) \mid \text{toint}(sterm) \]
\[ \text{indexof}(sterm, sterm) \]
\[ \text{lastindexof}(sterm, sterm) \]

\[ stern \rightarrow v \mid \varepsilon \mid s \]
\[ stern.sterm \mid stern|sterm \mid stern^* \mid (sterm) \]
\[ \text{charat}(sterm, iterm) \mid \text{tostring}(iterm) \]
\[ \text{toupper}(sterm) \mid \text{tolower}(sterm) \]
\[ \text{substring}(sterm, iterm, iterm) \]
\[ \text{replacefirst}(sterm, stern, stern) \]
\[ \text{replacelast}(sterm, stern, stern) \]
\[ \text{replaceall}(sterm, stern, stern) \]
Multi-track automata can also represent Presburger (linear arithmetic) arithmetic constraints

- Each track represents a single numeric variable
- Encoded as binary integers in 2’s complement form

\[
i = j \\
i \neq j \\
i = 2 \times j
\]
Linear Arithmetic Constraints

- Can be used to represent sets of valuations of unbounded integers
- Linear integer arithmetic formulas can be stored as a set of polyhedra

\[ F = \bigvee_k \bigwedge_l c_{kl} \]

where each \( c_{kl} \) is a linear equality or inequality constraint and each

\[ \bigwedge_l c_{kl} \]

is a polyhedron
Automata Representation for Arithmetic Constraints

[Bartzis, Bultan CIAA’02, IJFCS ’02]

- Given an atomic linear arithmetic constraint in one of the following two forms
  \[ \sum_{i=1}^{v} a_i \cdot x_i = c \]
  \[ \sum_{i=1}^{v} a_i \cdot x_i < c \]
  
  we construct a DFA which accepts all the solutions to the given constraint

- By combining such automata one can handle full Presburger arithmetic (linear arithmetic constraints + quantification)
Basic Construction

- We first construct a basic state machine which
  - Reads one bit of each variable at each step, starting from the least significant bits
  - and executes bitwise binary addition and stores the carry in each step in its state

Example

\[ x + 2y \]

\[
\begin{array}{c}
010 \\
+ 2 \times 001 \\
\hline
100
\end{array}
\]

Number of states: \( O\left(\sum_{i=1}^{v} |a_i|\right) \)
Automaton Construction

- **Equality With 0**
  - All transitions writing 1 go to a sink state
  - State labeled 0 is the only accepting state
  - For disequations (\(\neq\)), state labeled 0 is the only rejecting state

- **Inequality (<0)**
  - States with negative carries are accepting
  - No sink state

- **Non-zero Constant Term c**
  - Same as before, but now \(-c\) is the initial state
  - If there is no such state, create one (and possibly some intermediate states which can increase the size by \(|c|\))
Conjunction and Disjunction

Conjunction and disjunction is handled by generating the product automaton.

Automaton for $x-y<1$

Automaton for $2x-y>0$

Automaton for $x-y<1 \land 2x-y>0$
Integer Automata Construction

\( C \equiv x = -1 \land x + y = 1 \)
$C \equiv x = -1 \land x + y = 1$

$C_1 \equiv x + 0 \ast y + 1 = 0 \Rightarrow [1 \ 0 \ 1]$

$C_2 \equiv x + y - 1 = 0 \Rightarrow [1 \ 1 \ -1]$
**Integer Automata Construction**

\[ C \equiv x = -1 \land x + y = 1 \]
\[ C_1 \equiv x + 0 \times y + 1 = 0 \Rightarrow [1 \ 0 \ 1] \]
\[ C_2 \equiv x + y - 1 = 0 \Rightarrow [1 \ 1 \ -1] \]

- Using automata construction techniques described in:
Integer Automata Construction

\[ C \equiv x = -1 \land x + y = 1 \]

- Conjunction and disjunction is handled by automata product, negation is handled by automata complement

\[(111, 010) = (-1, 2)\]
Constraint Solving: Example Combining String and Integer Constraints

\[ i = 2 \times j \land \text{length}(v) = i \land \text{match}(v, (a \mid b)^*) \]

automaton for numeric variables
(v \_ auxiliary variable encoding length of v)

automaton for string variables
In general ABC constructs automata that over approximate the solution set

Some string constraints and combinations of string and integer constraints can lead to non-regular sets,

which means they are not representable as automata

ABC provides a sound over-approximation/abstraction:

If the automata does not accept any strings then the original formula is guaranteed to be NOT satisfiable

It is possible to also provide a sound under-approximation using automata
Model Counting String Constraints Solver

**INPUT**

string constraint: \( C \)

**Automata-Based model Counting string constraint solver (ABC)**

**OUTPUT**

counting function: \( f_c \)

length bound: \( k \)

\# of strings with length \( \leq k \) for which \( C \) evaluates to true

---

Aydin et al., Automata-based Model Counting for String Constraints. (CAV’15)
Can you solve it Will Hunting?

Given the graph

Find:
1) the adjacency matrix \( A \)
2) the matrix giving the number of 3 step walks
3) the generating function for walks from point \( i \to j \)
4) the generating function for walks from points \( 1 \to 3 \)
Automata-based Model Counting

- Converting constraints to automata reduces the model counting problem to path counting problem in graphs

\[ C \equiv \neg (x \in (01)^*) \]

- We will generate a function \( f(k) \)
  - Given length bound \( k \), it will count the number of paths with length \( k \).
  - \( f(0) = 0 \), \{\}\n  - \( f(1) = 2 \), \{0,1\}
  - \( f(2) = 3 \), \{00,10,11\}

We will generate a function \( f(k) \)

- Given length bound \( k \), it will count the number of paths with length \( k \).
- \( f(0) = 0 \), \{\}\n- \( f(1) = 2 \), \{0,1\}
- \( f(2) = 3 \), \{00,10,11\}
Path Counting via Matrix Exponentiation

\[ C = \neg (x \in (01)^*) \]

\[
T = \begin{bmatrix}
0 & 1 & 1 & 0 \\
1 & 0 & 1 & 1 \\
0 & 0 & 2 & 1 \\
0 & 0 & 0 & 0
\end{bmatrix},
T^2 = \begin{bmatrix}
1 & 0 & 3 & 2 \\
0 & 1 & 3 & 1 \\
0 & 0 & 4 & 2 \\
0 & 0 & 0 & 0
\end{bmatrix},
T^3 = \begin{bmatrix}
0 & 1 & 7 & 3 \\
1 & 0 & 7 & 4 \\
0 & 0 & 8 & 4 \\
0 & 0 & 0 & 0
\end{bmatrix},
T^4 = \begin{bmatrix}
0 & 1 & 15 & 8 \\
1 & 0 & 15 & 7 \\
0 & 0 & 16 & 8 \\
0 & 0 & 0 & 0
\end{bmatrix}
\]

\[ f(0) = 0 \quad f(1) = 2 \quad f(2) = 3 \quad f(3) = 8 \]
Path Counting via Recurrence Relation

\[ f(n, k) = \sum_{(m,n) \in E} f(m, k - 1) \]

- \( f(0,0) = 1 \)
- \( f(1,0) = 0 \)
- \( f(2,0) = 0 \)
- \( \ldots \)
- \( f(i, 0) = 0 \)
Path Counting via Recurrence Relation

\[
\begin{align*}
    f(4,k) &= f(2,k - 1) + f(3,k - 1) \\
    f(3,k) &= f(1,k - 1) + f(2,k - 1) + f(3,k - 1) \\
    f(2,k) &= f(1,k - 1) \\
    f(1,k) &= f(2,k - 1) \\
    f(1,0) &= 1, f(2,0) = 0, f(3,0) = 0, f(4,0) = 0
\end{align*}
\]
Path Counting via Recurrence Relation

- We can solve system of recurrence relations for final node

\[ f(0) = 0, \quad f(1) = 2, \quad f(2) = 3 \]

\[ f(k) = 2f(k-1) + f(k-2) - 2f(k-3) \]
Counting Paths via Generating Functions

- We can compute a generating function, \( g(z) \), for a DFA from the associated matrix.

\[
g(z) = (-1)^n \frac{\det(I - zT: n + 1,1)}{z \times \det(I - zT)} = \frac{2z - z^2}{1 - 2z - z^2 + 2z^3}
\]
Counting Paths via Generating Functions

\[ g(z) = \frac{2z - z^2}{1 - 2z - z^2 + 2z^3} \]

Each \( f(i) \) can be computed by Taylor expansion of \( g(z) \)

\[ g(z) = \frac{g(0)}{0!} z^0 + \frac{g^{(1)}(0)}{1!} z^1 + \frac{g^{(2)}(0)}{2!} z^2 + \cdots + \frac{g^{(n)}(0)}{n!} z^n + \cdots \]

\[ g(z) = 0z^0 + 2z^1 + 3z^2 + 8z^3 + 15z^4 + \cdots \]

\[ g(z) = f(0)z^0 + f(1)z^1 + f(2)z^2 + f(3)z^3 + f(4)z^4 + \cdots \]
Good job Will Hunting!

This is correct. Who did this?
Applicable to Both Automata

- Multi-track Binary Integer Automaton:

- String Automaton:
Model Counting String Constraints Solver

\[ C \rightarrow \text{Automata-Based model Counting string constraint solver (ABC)} \rightarrow f_C \rightarrow \text{counting function: length bound: } k \rightarrow \# \text{ of strings with length } \leq k \text{ for which } C \text{ evaluates to true} \]

Aydin et al., Automata-based Model Counting for String Constraints. (CAV'15)
Automata-based model counting extensions

- In order to scale the automata-based model counting, it is necessary to cache the prior results.

- Many constraints generated from programs are equivalent
  - By normalizing constraints we can identify many equivalent constraints.

- 87X improvement for the Kaluza big data set.
Kaluza Dataset:
1,342 big constraints and 17,554 small

1,342 big constraints are reduced to 34 equivalent constraints after normalization

17,554 small constraints are reduced to 360 equivalent constraints after normalization
Automata-based model counting extensions

- More caching
  - Cache subformulas
  - Automata provide a canonical form for constraints after minimization and determinization
  - Generate keys for automata and use a compute cache like BDDs
- Subformula caching leads to order of magnitude improvement for attack synthesis
ABC DEMO

https://github.com/vlab-cs-ucsb/ABC