Generic and Complete Techniques for Straight-Line String Constraints

Taolue Chen (Birkbeck)
Matthew Hague (Royal Holloway)
Anthony W. Lin (Kaiserslautern)
Philipp Ruemmer (Uppsala)
Zhilin Wu (Chinese Academy of Science)

Abstract

- New techniques for string constraint solving
  - Straight-line fragment
  - String operations/assertions not fixed
  - Two semantic-conditions (regularity)
  - Proof of decidability
- Implementation
  - OSTRICH solver
  - Competitive, expressive, and complete

String Programs

```
S  ::=   x := f(x1, ..., xn)
     |   assert g(x1, ..., xn)
     |   S1; S2
```

- f is a function from strings to strings
- g is a function from strings to boolean
- ; is sequential composition

Example

```
assert x in a*b*;
assert y in b*;
z := concat(x, y);
assert z in a*b*;
```

Example

assert in(x, a*b*)

assert x in a*b*;

assert y in b*;

z := concat(x, y)**;**

assert z in a*b*;

Example

assert in(x, a*b*)

```
assert x in a*b*;
assert y in b*;
z := concat(x, y);
assert z in a*b*;
```

Solution

Example

assert in(x, a*b*)

```
assert x in a*b*;
assert y in b*;
z := concat(x, y);
assert z in a*b*;
```

Solution

- x = aa

Example

assert in(x, a*b*)

```
assert x in a*b*;
assert y in b*;
z := concat(x, y);
assert z in a*b*;
```

Solution

- x = aa
- y = bb

Example

assert in(x, a*b*)

```
assert x in a*b*;
assert y in b*;
z := concat(x, y);
assert z in a*b*;
```

Solution

- x = aa
- y = bb
- (z = aabb)

Straight-Line Fragment

- Similar to single-static assignment form
  - Each variable only assigned once
  - Variables not used before they are assigned
    - Free-variables are never assigned
  - (Our language has no loop support)

Straight-Line Fragment

- Similar to single-static assignment form
  - Each variable only assigned once
  - Variables not used before they are assigned
    - Free-variables are never assigned
  - (Our language has no loop support)

Non-Example

```
x := concat(y,z)
y := x
y := z
```

Straight-Line Fragment

- Similar to single-static assignment form
  - Each variable only assigned once
  - Variables not used before they are assigned
    - Free-variables are neve
  - (Our language has no loo

Assigned after use
(Circular dependency)

Non-Example

```
x := concat(y,z)
y := x
y := z
```

Straight-Line Fragment

- Similar to single-static assignment form
  - Each variable only assigned once
  - Variables not used before they are assigned
    - Free-variables are neve
  - (Our language has no loo

Assigned after use
(Circular dependency)

Non-Example

```
x := concat(y,z)
y := x
y := z
```

Double assignment

Symbolic Execution

- Explore paths through a program
- Variables represented symbolically
- If-conditions &c. lead to constraints on variables
- Path is feasible if constraints are satisfiable
- Verification / Test-case generation
- Famous tools such as Klee

Example

Path

Program

```
function get_user_header(name)
    while name.contains("<script>")
        name = name.replaceAll("<script>", "")
    header = "<h1>" + name + "</h1>"
    assert not header.contains("script")
end
```

Example

Path

Program

```
function get_user_header(name)
  while name.contains("<script>")
     name = name.replaceAll("<script>", "")
  header = "<h1>" + name + "</h1>"
  assert not header.contains("script")
end
```

Example

Program

```
function get_user_header(name)
  while name.contains("<script>")
    name = name.replaceAll("<script>", "")
  header = "<h1>" + name + "</h1>"
  assert not header.contains("script")
end
```

Path

assert contains(n1, "<script>");

Example

Program

```
function get_user_header(name)
    while name.contains("<script>")
        name = name.replaceAll("<script>", "")
    header = "<h1>" + name + "</h1>"
    assert not header.contains("script")
end
```

Path

```
assert contains(n1, "<script>");

n2 := replaceAll(n1, "<script>", "");
```

Example

Program

```
function get_user_header(name)
  while name.contains("<script>")
    name = name.replaceAll("<script>", "")
  header = "<h1>" + name + "</h1>"
  assert not header.contains("script")
end
```

Path

```
assert contains(n1, "<script>");
n2 := replaceAll(n1, "<script>", "");
assert contains(n2, "<script>");
```

Example

Program

```
function get_user_header(name)
    while name.contains(“<script>”)
        name = name.replaceAll(“<script>”, “”)
    header = “<h1>” + name + “</h1>”
    assert not header.contains(“script”)
end
```

Path

```
assert contains(n1, “<script>”);
n2 := replaceAll(n1, “<script>”, “”);
assert contains(n2, “<script>”);
n3 := replaceAll(n2, “<script>”, “”);
```

Example

Program

```
function get_user_header(name)
  while name.contains("<script>")
    name = name.replaceAll("<script>", "")
  header = "<h1>" + name + "</h1>"
  assert not header.contains("script")
end
```

Path

assert contains(n1, "<script>");

n2 := replaceAll(n1, "<script>", "");

assert contains(n2, "<script>");

n3 := replaceAll(n2, "<script>", "");

assert not contains(n3, "<script>");

Example

Program

```
function get_user_header(name)
    while name.contains("<script>")
        name = name.replaceAll("<script>", "")
    header = "<h1>" + name + "</h1>"
    assert not header.contains("script")
end
```

Path

```
assert contains(n1, "<script>");
n2 := replaceAll(n1, "<script>", "");
assert contains(n2, "<script>");
n3 := replaceAll(n2, "<script>", "");
assert not contains(n3, "<script>");
hdr = concat("<h1>", n3, "</h1>");
```

Example

Program

```
function get_user_header(name)
    while name.contains("<script>")
        name = name.replaceAll("<script>", "")
    header = "<h1>" + name + "</h1>"
→   assert not header.contains("script")
end
```

Path

```
assert contains(n1, "<script>");
n2 := replaceAll(n1, "<script>", "");
assert contains(n2, "<script>");
n3 := replaceAll(n2, "<script>", "");
assert not contains(n3, "<script>");
hdr = concat("<h1>", n3, "</h1>");
assert contains(hdr, "<script>");
```

Example

Program

```
function get_user_header(name)
    while name.contains(“<script>”)
        name = name.replaceAll(“<script>”, “”)
    header = “<h1>” + name + “</h1>”
→   assert not header.contains(“script”)
end
```

Path

```
assert contains(n1, “<script>”);
n2 := replaceAll(n1, “<script>”, “”);
assert contains(n2, “<script>”);
n3 := replaceAll(n2, “<script>”, “”);
assert not contains(n3, “<script>”);
hdr = concat(“<h1>”, n3, “</h1>”);
assert contains(hdr, “<script>”);
```

Assertion in code negated

Example

Program

```
function get_user_header(name)
    while name.contains("<script>")
        name = name.replaceAll("<script>", "")
    header = "<h1>" + name + "</h1>"
→   assert not header.contains("script")
end
```

Path

```
assert contains(n1, "<script>");
n2 := replaceAll(n1, "<script>", "");
assert contains(n2, "<script>");
n3 := replaceAll(n2, "<script>", "");
assert not contains(n3, "<script>");
hdr = concat("<h1>", n3, "</h1>");
assert contains(hdr, "<script>");
```

Assertion in code negated

- No solution: path correct!

Solving Such Constraints

Straight-line with

- Regular constraints, concat, finite transductions
  - x := concat(y, z); x’ = T(x); **assert** x’ in a*b*;
  - EXPSPACE-c / PSPACE-c [Lin, Barcelo, 2016]
- Regular constraints, concat, replaceAll
  - x := replaceAll(y, e, z)
  - Undecidable if e can be a variable
  - EXPSPACE / PSPACE if e is a regular expression
  - Undecidable with length constraints
  - [Chen et al, 2018]

Generic Approach

Which string constraints can we allow?

- Maintain decidability
- Expressivity: capture most benchmarks
- Easy: solve with a straight-forward algorithm
- Extensible: allow users-defined string functions
- Efficient: solve competitively

Basic Approach: Go Backwards

For one variable, assume:

- **assert** g(x)
  - g is a regular constraint
- x : = f(y)
  - suppose x must satisfy a regular constraint
  - take the weakest precondition Pre(f, x)
  - Pre(f, x) is a regular constraint on y

Basic Approach: Go Backwards

For one variable, assume:

- **assert** g(x)
  - g is a regular constraint
- x : = f(y)
  - suppose x must satisfy a regular constraint
  - take the weakest precondition Pre(f, x)
  - Pre(f, x) is a regular constraint on y

Regular contraints on output variables become regular constraints on input variables.

Example

```
assert x in a*b*;
y = reverse(x);
assert y in b*a*;
z = replaceAll(y, a, b);
assert z in b*;
```

Example

```
assert x in a*b*;
y = reverse(x);
assert y in b*a*;
z = replaceAll(y, a, b);
assert z in b*;
```

} **assert** y in (a | b)*;

Example

assert x in a*b*;

y = reverse(x);

assert y in b*a*;

assert y in (a | b)*;

Example

assert x in a*b*;

y = reverse(x);

assert y in b*a*;

assert y in (a | b)*;

} **assert** y in (a | b)* & b*a*;

Example

```
assert x in a*b*;
y = reverse(x);
assert y in (a | b)* & b*a*;
```

Example

assert x in a*b*;

y = reverse(x);

assert y in (a | b)* & b*a*;

} **assert** x in a*b*;

Example

assert x in a*b*;

assert x in a*b*;

Example

assert x in a*b*;

assert x in a*b*;

} **assert** x in a*b*;

Example

assert x in a*b*;

Example

Easy to solve

assert x in a*b*;

Algorithm in General

Assertions and functions may take several variables

- **assert** g(x1, ..., xn)
  - g admits a regular monadic decomposition
  - i.e. ⋃ L1 x ... x Ln
- x := f(x1, ..., xn)
  - if x is a regular language, then
  - Pre(f, x) is ⋃ L1 x ... x Ln

Algorithm in General

Assertions and functions may take several variables

- **assert** g(x1, ..., xn)
  - g admits a regular monadic decomposition
  - i.e. $\bigcup$ L1 x ... x Ln
- x := f(x1, ..., xn)
  - if x is a regular language, then
  - Pre(f, x) is $\bigcup$ L1 x ... x Ln

Given these, the backwards algorithm still works

Genericity

Which string functions satisfy these constraints?

- Concatenation
- Reverse
- One-way / Two-way transductions
- x := replaceAll(y, e, z)

Subsume previous results and allow extensions

- E.g. capture groups in real-world regular expressions

Complexity

Depends on string operations permitted

- PSPACE – conjunction of regular constraints
- EXPSPACE – concat, one-way transductions, replaceAll
- Non-elementary – two-way non-deterministic transductions
- Undecidable – equals(x, y) and replaceAll(x, a, y)

Complexity

Depends on string operations permitted

- PSPACE – conjunction of regular constraints
- EXPSPACE – concat, one-way transductions, replaceAll
- Non-elementary – two-way non-deterministic transductions
- Undecidable – equals(x, y) and replaceAll(x, a, y)

Determinism handled carefully

Complexity

Depends on string operations permitted

- PSPACE – conjunction of regular constraints
- EXPSPACE – concat, one-way transductions, replaceAll
- Non-elementary – two-way non-deterministic transductions
- Undecidable – equals(x, y) and replaceAll(x, a, y)

Determinism handled carefully

- f^{-1}(L1 & L2) = f^{-1}(L1) & f^{-1}(L2) if f deterministic

Complexity

Depends on string operations permitted

- PSPACE – conjunction of regular constraints
- EXPSPACE – concat, one-way transductions, replaceAll
- Non-elementary – two-way non-deterministic transductions
- Undecidable – equals(x, y) and replaceAll(x, a, y)

Determinism handled carefully

- $f^{-1}(L1 \& L2) = f^{-1}(L1) \& f^{-1}(L2)$ if f deterministic
- avoid taking conjunctions until the end

OSTRICH

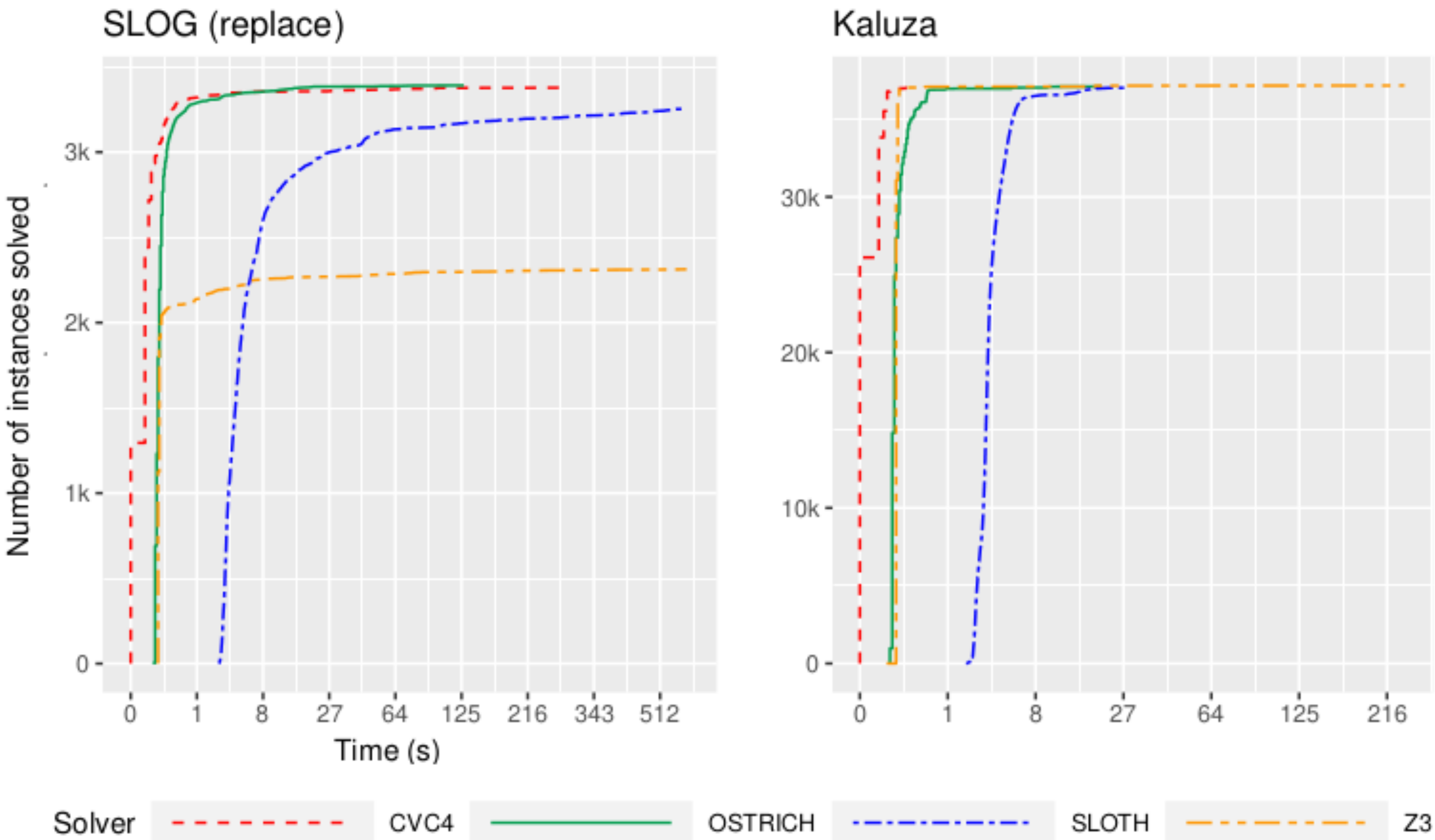
Approach implemented in OSTRICH

- Written in Scala
- Built on Princess SMT solver
- Extensible
  - Each string operation is a single class
  - New operations easily added

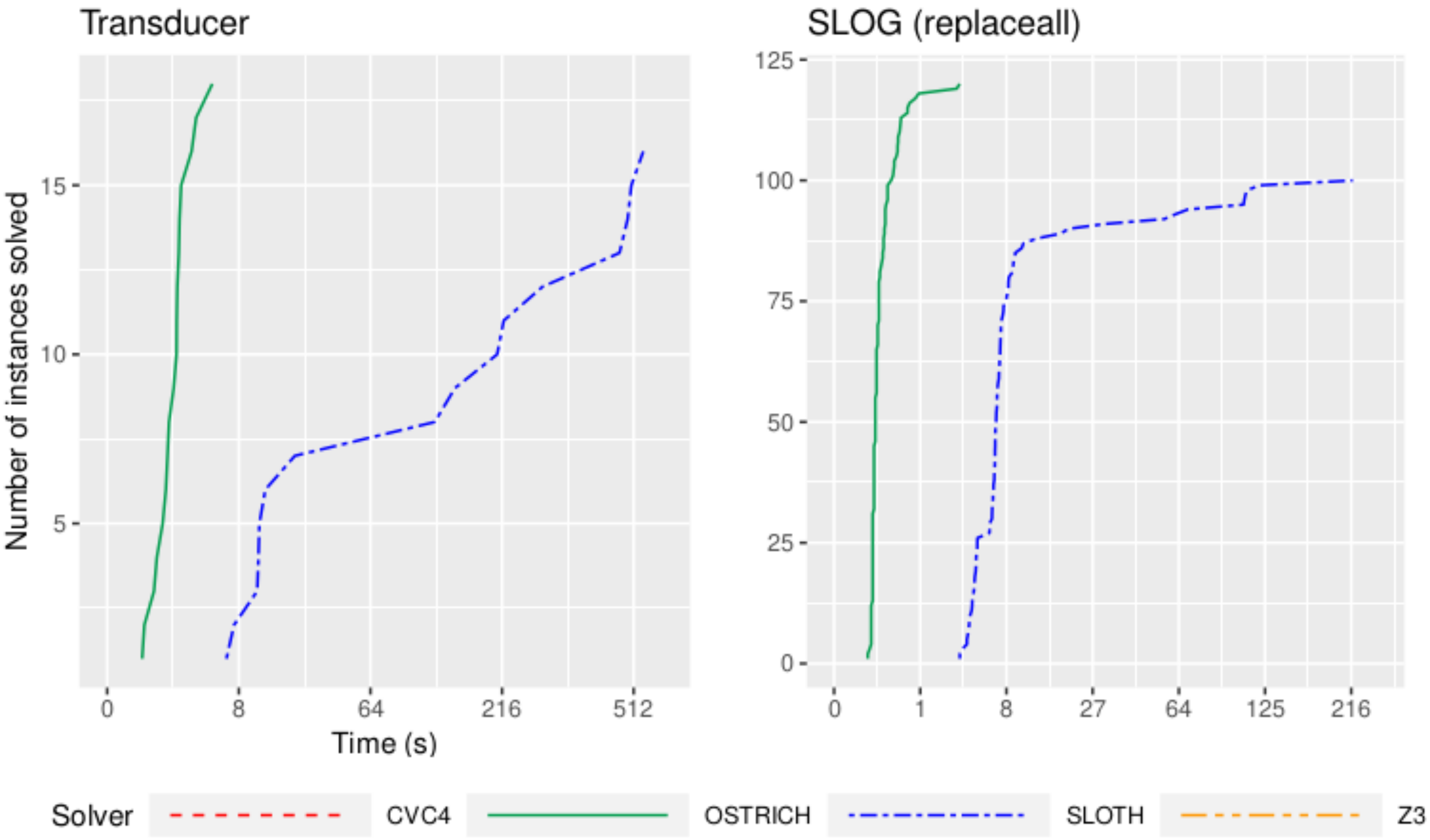
Benchmarking

- Kaluza, Stranger, SLOG examples
- Compared with CVC 4.1.6, Z3-str, and SLOTH

Benchmarks on All Solvers



Benchmarks Unique Features



Optimisations

Pre-image computation should be done carefully

- x := concat(y, z)
- Pre(concat, L) = ∪ Lq x qL
  - Lq – words to state q
  - qL – word from state q
- Multiplies search by number of states
- Only choose q that are feasible

Pre-image of replaceAll uses Caley graphs

Summary

- Generic decision procedure for straight-line string constriants
- Semantic conditions for decidability
  - Regular monadic decomposition
- OSTRICH
  - Competitive on popular benchmarks
  - Extensible with new string operations