Translation of Semi-Extended Regular Expressions using ~~Derivatives~~ Linear Forms

Antoine Martin[1], Etienne Renault[2], Alexandre Duret-Lutz[1]

[1]LRE (EPITA), [2]SiPearl

Séminaire d'informatique théorique, Rouen — 2024-10-17

Martin, Renault, and Duret-Lutz. **Translation of semi-extended regular expressions using derivatives.** *CIAA'24.* doi

Context: SVA & PSL

Industrial standards for the specification of electronic systems, both born in 2005.

SystemVerilog Assertions	Property Specification Language
Part of the SystemVerilog language.	Not tied to a particular tool. Has "*flavors*."

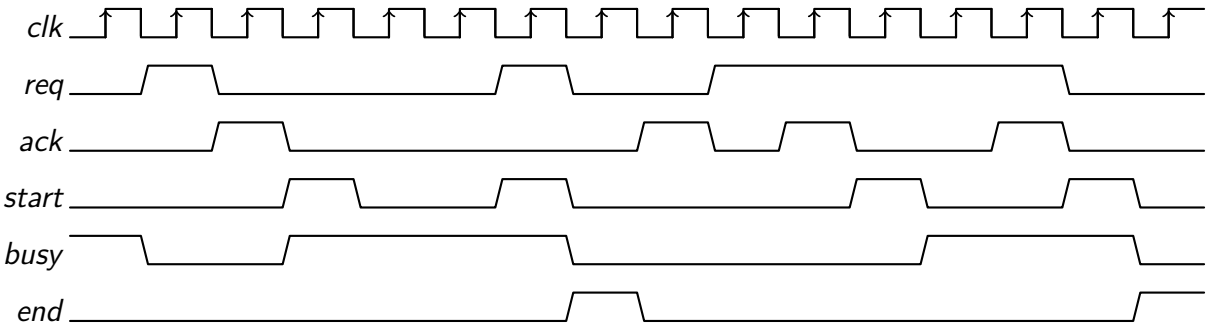
- Both are linear temporal logics built over a sub-language of "regular expressions":
  - "Sequences" in SVA
  - "Sequential Extended Regular Expressions" (SERE) in PSL
- Higher operators can combine these SERE, and also include traditional LTL operators.
- As expressive as ω-regular languages (unlike LTL).


1850-2010 - IEEE Standard for Property Specification Language (PSL). IEEE, Apr. 2010. doi


1800-2023 - IEEE Standard for SystemVerilog–Unified Hardware Design, Specification, and Verification Language. IEEE, Feb. 2024. doi

Context: Specification over Multiple Boolean Signals

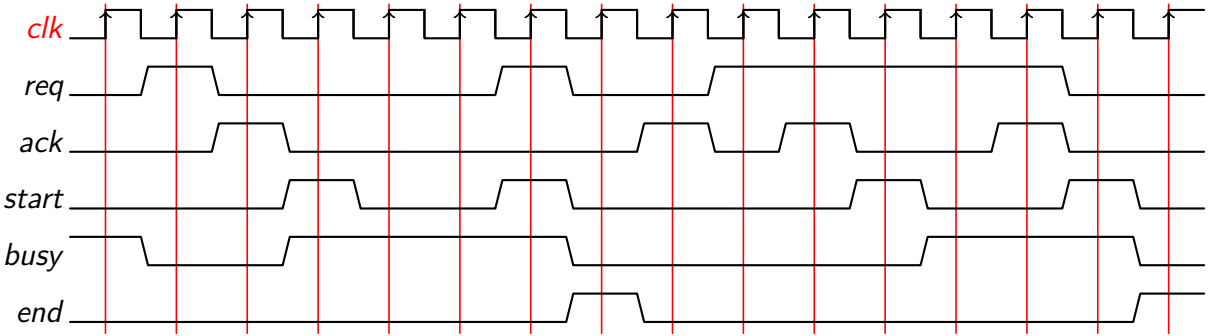
`assert` ({`1[*]` ; *req* ; *ack*}`[]=>`{*start* ; *busy*`[*]` ; *end*}`!`)`@`(`posedge` *clk*)



Context: Specification over Multiple Boolean Signals

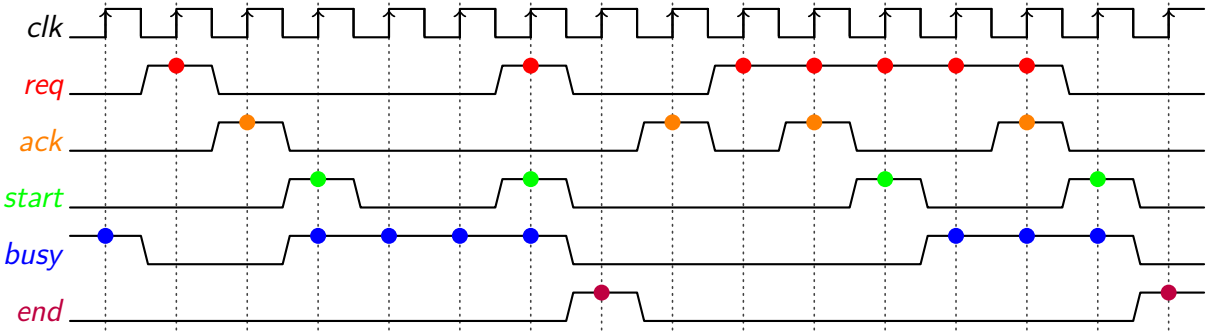
discrete time

`assert ({1[*] ; req ; ack}[]=>{start ; busy[*] ; end}!)@(posedge clk)`



Context: Specification over Multiple Boolean Signals

`assert ({1[*] ; req ; ack}[]=>{start ; busy[*] ; end}!)@(posedge clk)`

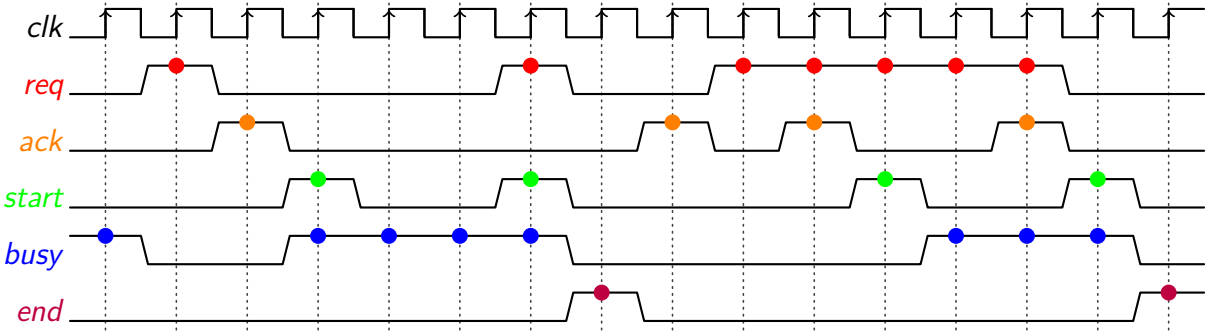


Contex[illegible]n c[illegible]an Signals

any prefix matching this SERE

must precede a sequence matching this SERE

```
assert ({1[*] ; req ; ack}[]=>{start ; busy[*] ; end}!)@(posedge clk)
```

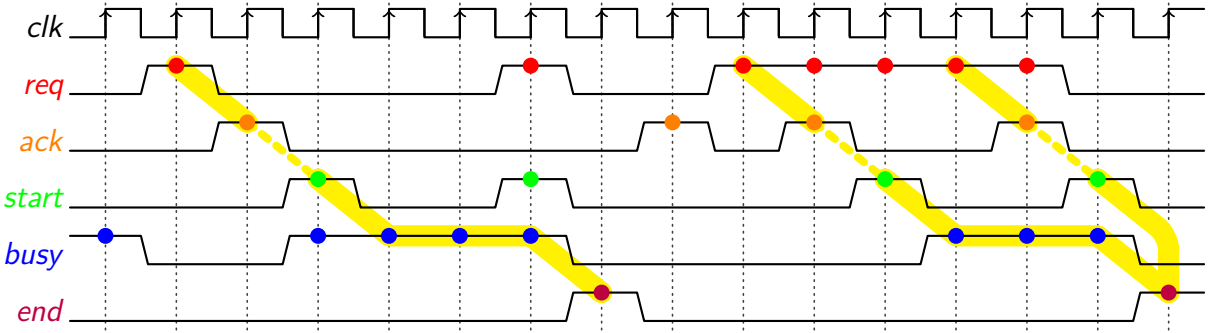


Contex[illegible]n [illegible]an Signals

any prefix matching this SERE

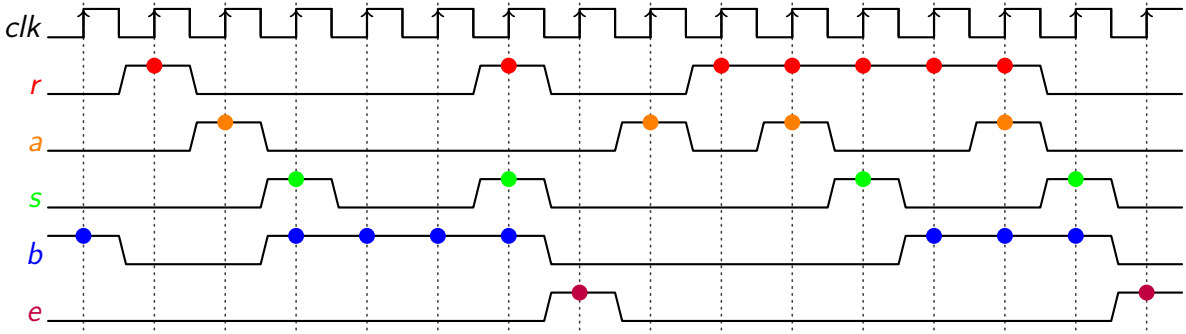
must precede a sequence matching this SERE

```
assert ({1[*] ; req ; ack}[]=>{start ; busy[*] ; end}!)@(posedge clk)
```



Context: Specification over Multiple Boolean Signals

`assert ({1[*] ; r ; a}[]=>{s ; b[*] ; e}!)@(posedge clk)`



$\underbrace{\bar{r}\bar{a}\bar{s}b\bar{e}}; \underbrace{r\bar{a}\bar{s}\bar{b}\bar{e}}; \underbrace{\bar{r}a\bar{s}\bar{b}\bar{e}}; \underbrace{\bar{r}\bar{a}sb\bar{e}}; \underbrace{\bar{r}\bar{a}\bar{s}b\bar{e}}; \ldots$

interpretation as a word over $\Sigma = 2^{\{r,a,s,b,e\}}$

Context: Converting PSL/SVA to Büchi Automata

$\{a[->2]\}!\&\&(\{a[->1] : b\}[]=>\{a[->1] : c\})\&\&(\{a[->1] : !\, b\}[]=>\{a[->1] : !\, c\})$

How to convert a PSL/SVA formula into a Büchi automaton?

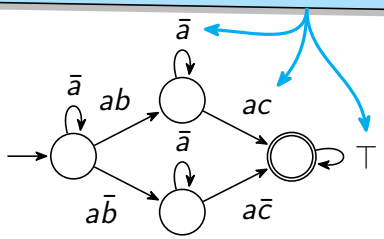
1. Convert SEREs into DFAs/NFAs (this talk!)
2. Combine those automata...

Boolean formulas as abbreviations:

$\bar{a} = \{\bar{a}\bar{b}\bar{c}, \bar{a}\bar{b}c, \bar{a}b\bar{c}, \bar{a}bc\}$

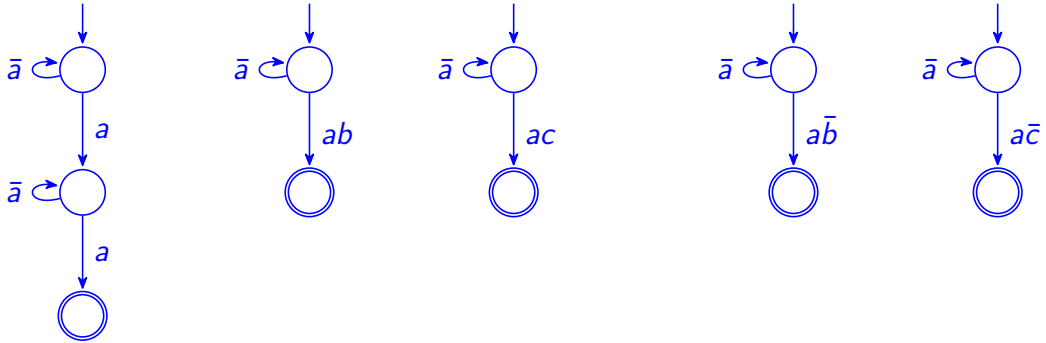
$ab = \{ab\bar{c}, abc\}$

$\top = 2^{\{a,b,c\}}$



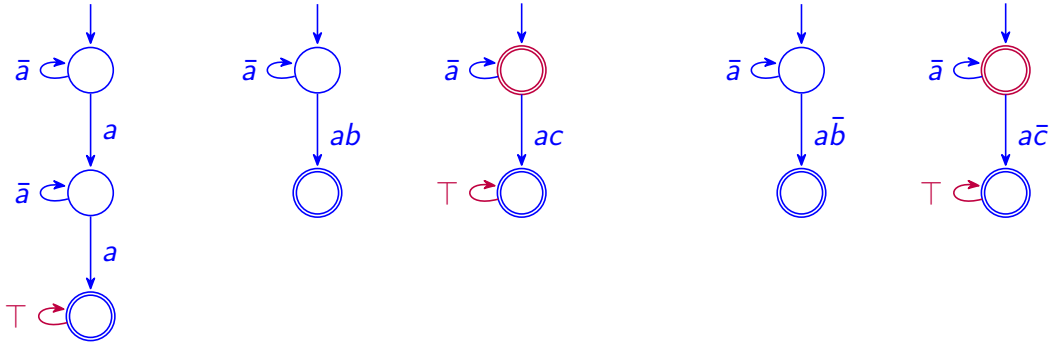
Context: Converting PSL/SVA to Büchi Automata

$\{a[->2]\}!\&\&(\{a[->1] : b\}[]=>\{a[->1] : c\})\&\&(\{a[->1] : !\, b\}[]=>\{a[->1] : !\, c\})$



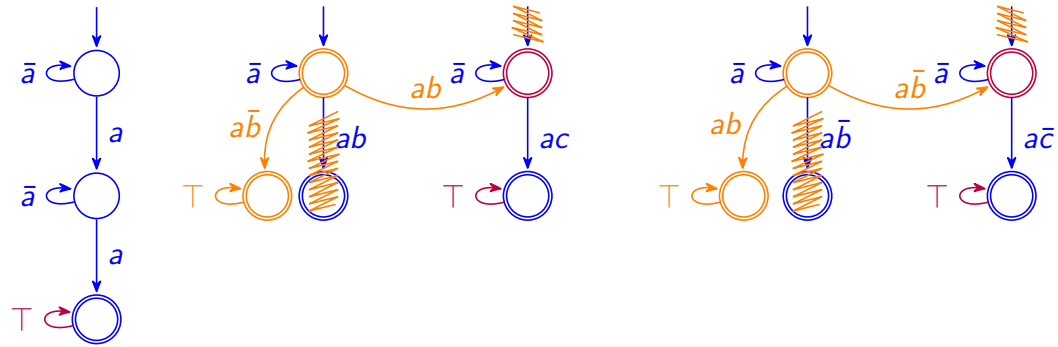
Context: Converting PSL/SVA to Büchi Automata

$\{a[->2]\}!\&\&(\{a[->1] : b\}[]=>\{a[->1] : c\})\&\&(\{a[->1] : !\, b\}[]=>\{a[->1] : !\, c\})$

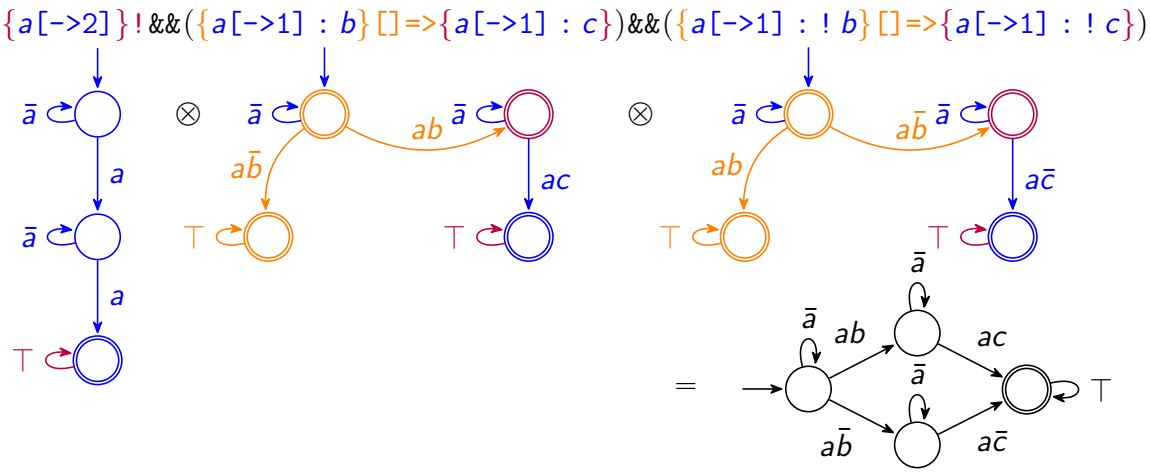


Context: Converting PSL/SVA to Büchi Automata

$\{a[->2]\}!\&\&(\{a[->1] : b\}[]=>\{a[->1] : c\})\&\&(\{a[->1] : !\, b\}[]=>\{a[->1] : !\, c\})$



Context: Converting PSL/SVA to Büchi Automata



Back to our Subject: What's a SERE?

Boolean formulas over signals

- AP is the set of "atomic propositions",
- $\Sigma = 2^{AP}$ is our alphabet
- Boolean formulas denote sets of letters:

$$a \wedge b = \{abc,\ ab\bar{c}\}$$

Challenges

- Σ can be large
- We want automata labeled by Boolean formulas (to keep them small)
- We have to support more than the usual regular operators.

SERE operators

Regular operators:

- Concatenation: e_1 ; e_2
- Choice: e_1 || e_2
- Kleene star: e[*]

Additional operators:

- Intersection: e_1 && e_2
- First match: first_match(e)
- Fusion: e_1 : e_2

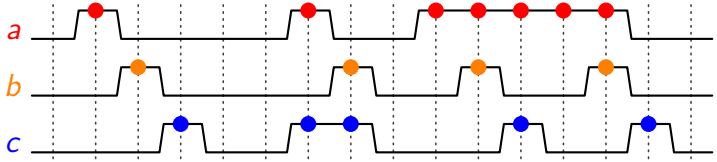
A lot of syntactic sugar:

- e[+], e[->n], e[=n], e_1 & e_2, ...

Fusion and First match

Count the number of prefixes that match the following SEREs:

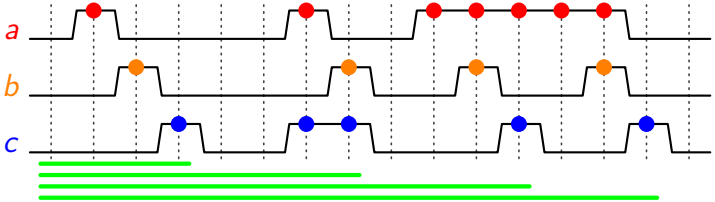
- `1[*]` ; $(a$ `||` $b)$; c



Fusion and First match

Count the number of prefixes that match the following SEREs:

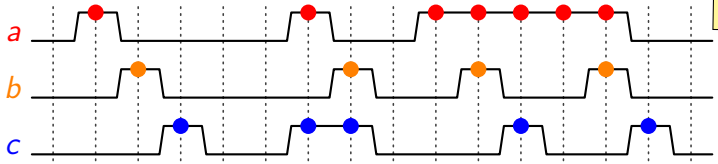
- `1[*] ; (a || b) ; c` 4 matches



Fusion and First match

Count the number of prefixes that match the following SEREs:

- `1[*]` ; $(a$ `||` $b)$; c 4 matches
- `first_match`$(1[*]$; $(a$ `||` $b)$; $c)$



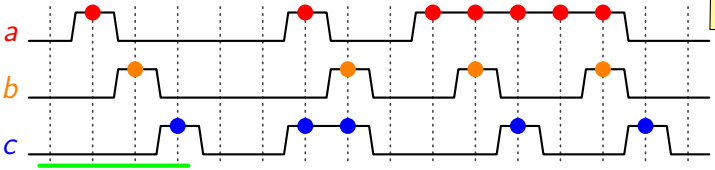
One intuition for `first_match`

In a DFA for SERE e, remove the outgoing transitions of all accepting states to obtain a DFA for `first_match`(e).

Fusion and First match

Count the number of prefixes that match the following SEREs:

- $1[*] ; (a \,||\, b) ; c$ 4 matches
- $\texttt{first\_match}(1[*] ; (a \,||\, b) ; c)$ 1 match



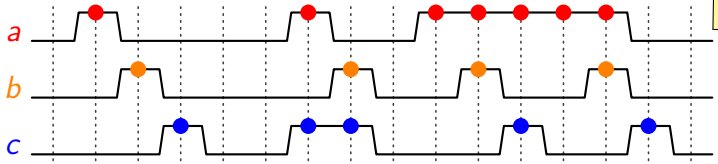
One intuition for `first_match`

In a DFA for SERE e, remove the outgoing transitions of all accepting states to obtain a DFA for $\texttt{first\_match}(e)$.

Fusion and First match

Count the number of prefixes that match the following SEREs:

- $1[*] ; (a \,||\, b) ; c$ 4 matches
- $\texttt{first\_match}(1[*] ; (a \,||\, b) ; c)$ 1 match
- $\texttt{first\_match}(1[*] ; (a \,||\, b)) ; c$



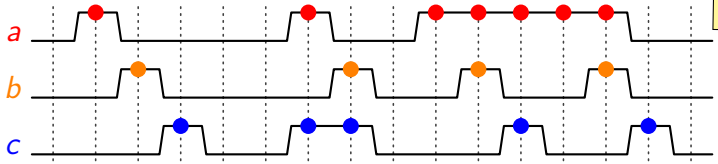
One intuition for `first_match`

In a DFA for SERE e, remove the outgoing transitions of all accepting states to obtain a DFA for $\texttt{first\_match}(e)$.

Fusion and First match

Count the number of prefixes that match the following SEREs:

- `1[*]` ; $(a$ `||` $b)$; c — 4 matches
- `first_match`$($`1[*]` ; $(a$ `||` $b)$; $c)$ — 1 match
- `first_match`$($`1[*]` ; $(a$ `||` $b))$; c — 0 match



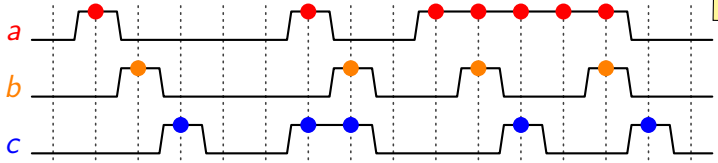
One intuition for `first_match`

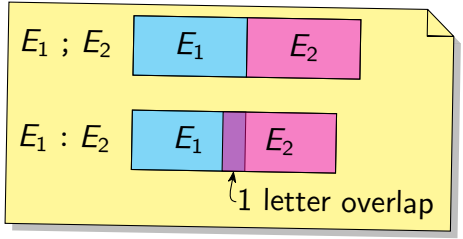
In a DFA for SERE e, remove the outgoing transitions of all accepting states to obtain a DFA for `first_match`(e).

Fusion and First match

Count the number of prefixes that match the following SEREs:

- `1[*]` ; $(a$ `||` $b)$; c — 4 matches
- `first_match`$(1$`[*]` ; $(a$ `||` $b)$; $c)$ — 1 match
- `first_match`$(1$`[*]` ; $(a$ `||` $b))$; c — 0 match
- $(1$`[*]` ; a ; b`[->2]`$)$: $(a$`[*2]`$)$

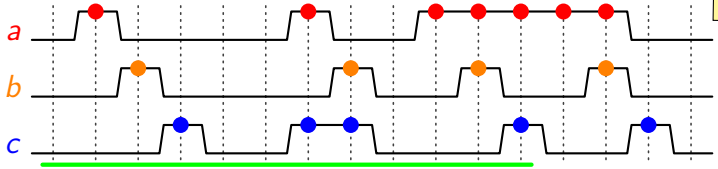


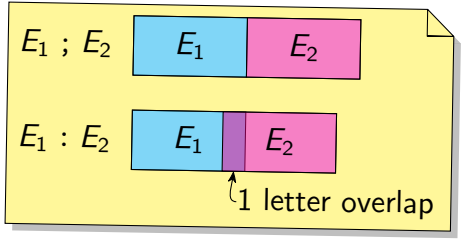


Fusion and First match

Count the number of prefixes that match the following SEREs:

- `1[*]` ; $(a$ `||` $b)$; c — 4 matches
- `first_match`$(1$`[*]` ; $(a$ `||` $b)$; $c)$ — 1 match
- `first_match`$(1$`[*]` ; $(a$ `||` $b))$; c — 0 match
- $(1$`[*]` ; a ; b`[->2]`$)$: $(a$`[*2]`$)$ — 1 match





Classical RE-to-Automata Translations

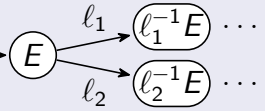
Given a RE E, and letter $\ell \in \Sigma$.

Brzozowski's derivatives

$\ell^{-1}E$ is a RE, such that
$\mathscr{L}(\ell^{-1}E) = \{w \mid \ell w \in \mathscr{L}(E)\}$.

$$a^{-1}(ab + a)^\star = (b + \varepsilon)(ab + a)^\star$$

Build a DFA:



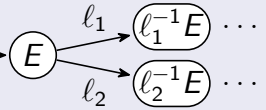
Classical RE-to-Automata Translations

Given a RE E, and letter $\ell \in \Sigma$.

Brzozowski's derivatives

$\ell^{-1}E$ is a RE, such that
$\mathscr{L}(\ell^{-1}E) = \{w \mid \ell w \in \mathscr{L}(E)\}$.

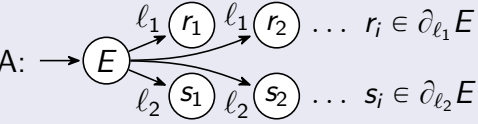
$a^{-1}(ab + a)^\star = (b + \varepsilon)(ab + a)^\star$

Build a DFA: 

Antimirov's partial-derivatives

$\partial_\ell E$ is **a set** of REs such that
$\bigcup_{r \in \partial_\ell E} \mathscr{L}(r) = \{w \mid \ell w \in \mathscr{L}(E)\}$.

$\partial_a(ab + a)^\star = \{b(ab + a)^\star, (ab + a)^\star\}$

NFA: 

Antimirov. **Partial derivatives of regular expressions and finite automaton constructions.** *TCS*, Mar. 1996. doi

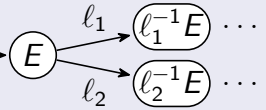
Classical RE-to-Automata Translations

Given a RE E, and letter $\ell \in \Sigma$.

Brzozowski's derivatives

$\ell^{-1}E$ is a RE, such that
$\mathscr{L}(\ell^{-1}E) = \{w \mid \ell w \in \mathscr{L}(E)\}$.

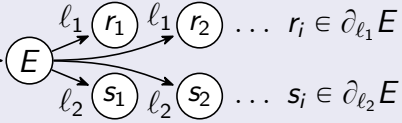
$$a^{-1}(ab + a)^\star = (b + \varepsilon)(ab + a)^\star$$

Build a DFA: 

Antimirov's partial-derivatives

$\partial_\ell E$ is **a set** of REs such that
$\bigcup_{r \in \partial_\ell E} \mathscr{L}(r) = \{w \mid \ell w \in \mathscr{L}(E)\}$.

$$\partial_a(ab + a)^\star = \{b(ab + a)^\star, (ab + a)^\star\}$$

NFA: 

Linear Forms

$$\mathrm{LF}(E) = \{(\ell_1, r_1), (\ell_1, r_2), \ldots (\ell_2, s_1), (\ell_2, s_2), \ldots\}$$

Antimirov. **Partial derivatives of regular expressions and finite automaton constructions.** *TCS*, Mar. 1996. doi

Classical R

3. The components $\tau(P, x)$ of the transition function should be computed through the function $lf(\_)$ which gives a *whole tuple* of transitions $\{\langle x, \partial_x(P)\rangle \mid x \in \mathscr{A}\}$ in one pass over P. This is more efficient than to compute separately each derivative w.r.t. $x \in \mathscr{A}$ that requires one pass over P for each x. The bigger the alphabet, the more one gains from this optimization.

Therefore, our modification can be implemented much more efficiently than the original Brzozowski construction.[9]

Antimirov (1996)

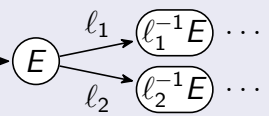
Given a RE E

Brzozowsk

$\ell^{-1}E$ is a RE, such that
$\mathscr{L}(\ell^{-1}E) = \{w \mid \ell w \in \mathscr{L}(E)\}$.

$$a^{-1}(ab + a)^\star = (b + \varepsilon)(ab + a)^\star$$

Build a DFA:



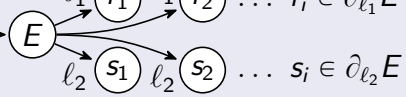
vatives

at

$(E)\}$.

$$\partial_a(ab + a)^\star = \{b(ab + a)^\star, (ab + a)^\star\}$$

NFA:



Linear Forms

$$\mathrm{LF}(E) = \{(\ell_1, r_1), (\ell_1, r_2), \ldots (\ell_2, s_1), (\ell_2, s_2), \ldots\}$$

Antimirov. **Partial derivatives of regular expressions and finite automaton constructions.** *TCS*, Mar. 1996. doi

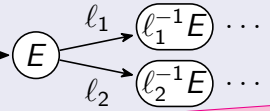
Classical RE-to-Automata Translations

Given a RE E, and letter $\ell \in \Sigma$.

Brzozowski's derivatives

$\ell^{-1}E$ is a RE, such that
$\mathscr{L}(\ell^{-1}E) = \{w \mid \ell w \in \mathscr{L}(E)\}$.

$a^{-1}(ab + a)^\star = (b + \varepsilon)(ab + a)^\star$

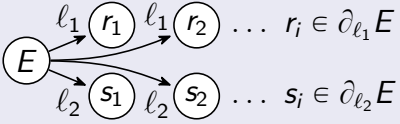
Build a DFA: 

When $\Sigma = 2^{AP}$, we want
$\mathrm{LF}(E) = \{\ldots, (b_i, r_i) \ldots\}$
where b_i is a Boolean formula.

Antimirov's partial-derivatives

$\partial_\ell E$ is **a set** of REs such that
$\bigcup_{r \in \partial_\ell E} \mathscr{L}(r) = \{w \mid \ell w \in \mathscr{L}(E)\}$.

$\partial_a(ab + a)^\star = \{b(ab + a)^\star, (ab + a)^\star\}$

NFA: 

Linear Forms

$$\mathrm{LF}(E) = \{(\ell_1, r_1), (\ell_1, r_2), \ldots (\ell_2, s_1), (\ell_2, s_2), \ldots\}$$

Antimirov's Linear Forms

$$
\begin{aligned}
\mathsf{LF}(\bot) &= \varnothing \\
\mathsf{LF}(\varepsilon) &= \varnothing \\
\mathsf{LF}(\ell) &= \{(\ell, \varepsilon)\} \\
\mathsf{LF}(r_1 \vee r_2) &= \mathsf{LF}(r_1) \cup \mathsf{LF}(r_2) \\
\mathsf{LF}(r^\star) &= \mathsf{LF}(r) \mathbin{.} r^\star \\
\mathsf{LF}(r_1 \mathbin{.} r_2) &= (\mathsf{LF}(r_1) \mathbin{.} r_2) \cup (\lambda(r_1) \mathbin{.} \mathsf{LF}(r_2))
\end{aligned}
$$

Antimirov's Linear Forms

$$
\begin{aligned}
\mathsf{LF}(\bot) &= \varnothing \\
\mathsf{LF}(\varepsilon) &= \varnothing \\
\mathsf{LF}(\ell) &= \{(\ell, \varepsilon)\} \\
\mathsf{LF}(r_1 \vee r_2) &= \mathsf{LF}(r_1) \cup \mathsf{LF}(r_2) \\
\mathsf{LF}(r^\star) &= \mathsf{LF}(r) \mathbin{.} r^\star \\
\mathsf{LF}(r_1 \mathbin{.} r_2) &= (\mathsf{LF}(r_1) \mathbin{.} r_2) \cup (\lambda(r_1) \mathbin{.} \mathsf{LF}(r_2))
\end{aligned}
$$

Linear Forms for SEREs

$$
\begin{aligned}
\mathrm{LF}(0) &= \varnothing \\
\mathrm{LF}([*0]) &= \varnothing \\
\mathrm{LF}(b) &= \{(b, \varepsilon)\} \\
\mathrm{LF}(r_1 \mid\mid r_2) &= \mathrm{LF}(r_1) \cup \mathrm{LF}(r_2) \\
\mathrm{LF}(r\,[*]) &= \mathrm{LF}(r)\,;\, r\,[*] \\
\mathrm{LF}(r_1\,;\, r_2) &= (\mathrm{LF}(r_1)\,;\, r_2) \cup (\lambda(r_1)\,;\, \mathrm{LF}(r_2))
\end{aligned}
$$

Linear Forms for SEREs

$$
\begin{aligned}
\mathsf{LF}(0) &= \varnothing \\
\mathsf{LF}(\,[*0]\,) &= \varnothing \\
\mathsf{LF}(b) &= \{(b, \varepsilon)\} \\
\mathsf{LF}(r_1 \mid\mid r_2) &= \mathsf{LF}(r_1) \cup \mathsf{LF}(r_2) \\
\mathsf{LF}(r\,[*]\,) &= \mathsf{LF}(r)\,;\, r\,[*] \\
\mathsf{LF}(r_1\,;\, r_2) &= (\mathsf{LF}(r_1)\,;\, r_2) \cup (\lambda(r_1)\,;\, \mathsf{LF}(r_2)) \\
\mathsf{LF}(r_1 : r_2) &= (\mathsf{LF}(r_1) : r_2) \cup \left\{ (p_i \wedge p_j, s_j) \;\middle|\; \begin{array}{l} (p_i, s_i) \in \mathsf{LF}(r_1),\ \lambda(s_i) = \varepsilon, \\ (p_j, s_j) \in \mathsf{LF}(r_2) \end{array} \right\} \\
\mathsf{LF}(r_1 \,\&\&\, r_2) &= \{(p_i \wedge p_j, s_i \,\&\&\, s_j) \mid (p_i, s_i) \in \mathsf{LF}(r_1),\ (p_j, s_j) \in \mathsf{LF}(r_2)\} \\
\mathsf{LF}(\texttt{first\_match}(r)) &= \{(p_i, \texttt{first\_match}(s_i)) \mid (p_i, s_i) \in \det(\mathsf{LF}(r))\}
\end{aligned}
$$

Algorithm 1: Translation

input : A SERE φ
output: An automaton $\mathcal{A}$ such that
$\mathscr{L}(\mathcal{A}) = \mathscr{L}(\varphi)$

$Q, \delta, F \leftarrow \{\varphi\}, \varnothing, \varnothing$;
todo.push(φ);
while todo $\neq \varnothing$ **do**
  $f \leftarrow$ todo.pop();
  foreach $(p, s) \in \mathrm{LF}(f)$ **do**
    if $s \notin Q$ **then**
      $Q \leftarrow Q \cup \{s\}$;
      todo.push (s);
      if $\varepsilon \models s$ **then**
        $F \leftarrow F \cup \{s\}$;
    $\delta \leftarrow \delta \cup \{f \xrightarrow{p} s\}$;
return $\langle Q, \delta, \varphi, F \rangle$;

Algorithm 1: Translation

input : A SERE φ
output: An automaton $\mathcal{A}$ such that
$\mathscr{L}(\mathcal{A}) = \mathscr{L}(\varphi)$

$Q, \delta, F \leftarrow \{\varphi\}, \varnothing, \varnothing$;
todo.push(φ);
while todo $\neq \varnothing$ **do**
  $f \leftarrow$ todo.pop();
  foreach $(p, s) \in \mathrm{LF}(f)$ **do**
    if $s \notin Q$ **then**
      $Q \leftarrow Q \cup \{s\}$;
      todo.push (s);
      if $\varepsilon \models s$ **then**
        $F \leftarrow F \cup \{s\}$;
    $\delta \leftarrow \delta \cup \{f \xrightarrow{p} s\}$;
return $\langle Q, \delta, \varphi, F \rangle$;

Initial state is the input formula φ

Algorithm 1: Translation

input : A SERE φ
output: An automaton $\mathcal{A}$ such that
$\mathscr{L}(\mathcal{A}) = \mathscr{L}(\varphi)$

$Q, \delta, F \leftarrow \{\varphi\}, \varnothing, \varnothing$;
todo.push(φ);
while todo $\neq \varnothing$ **do**
  $f \leftarrow$ todo.pop();
  foreach $(p, s) \in \mathrm{LF}(f)$ **do**
    if $s \notin Q$ **then**
      $Q \leftarrow Q \cup \{s\}$;
      todo.push (s);
      if $\varepsilon \models s$ **then**
        $F \leftarrow F \cup \{s\}$;
    $\delta \leftarrow \delta \cup \{f \xrightarrow{p} s\}$;
return $\langle Q, \delta, \varphi, F \rangle$;

Initial state is the input formula φ

Loop over LF pairs

Algorithm 1: Translation

input : A SERE φ
output: An automaton $\mathcal{A}$ such that
$\mathscr{L}(\mathcal{A}) = \mathscr{L}(\varphi)$

$Q, \delta, F \leftarrow \{\varphi\}, \varnothing, \varnothing$;
todo.push(φ);
while todo $\neq \varnothing$ **do**
  $f \leftarrow$ todo.pop();
  foreach $(p, s) \in \mathrm{LF}(f)$ **do**
    if $s \notin Q$ **then**
      $Q \leftarrow Q \cup \{s\}$;
      todo.push (s);
      if $\varepsilon \models s$ **then**
        $F \leftarrow F \cup \{s\}$;
    $\delta \leftarrow \delta \cup \{f \xrightarrow{p} s\}$;
return $\langle Q, \delta, \varphi, F \rangle$;

Initial state is the input formula φ

Loop over LF pairs

States are labeled by SEREs

Algorithm 1: Translation

input : A SERE φ
output: An automaton $\mathcal{A}$ such that
$\mathscr{L}(\mathcal{A}) = \mathscr{L}(\varphi)$

$Q, \delta, F \leftarrow \{\varphi\}, \varnothing, \varnothing$;
todo.push(φ);
while todo $\neq \varnothing$ **do**
  $f \leftarrow$ todo.pop();
  foreach $(p, s) \in \mathrm{LF}(f)$ **do**
    if $s \notin Q$ **then**
      $Q \leftarrow Q \cup \{s\}$;
      todo.push (s);
      if $\varepsilon \models s$ **then**
        $F \leftarrow F \cup \{s\}$;
    $\delta \leftarrow \delta \cup \{f \xrightarrow{p} s\}$;
return $\langle Q, \delta, \varphi, F \rangle$;

Initial state is the input formula φ

Loop over LF pairs

States are labeled by SEREs

Identify accepting states

Algorithm 1: Translation

input : A SERE φ
output: An automaton $\mathcal{A}$ such that
$\mathscr{L}(\mathcal{A}) = \mathscr{L}(\varphi)$

$Q, \delta, F \leftarrow \{\varphi\}, \varnothing, \varnothing$;
todo.push(φ);
while todo $\neq \varnothing$ **do**
  $f \leftarrow$ todo.pop();
  foreach $(p, s) \in \boxed{\mathrm{LF}(f)}$ **do**
    if $s \notin Q$ **then**
      $Q \leftarrow Q \cup \{s\}$;
      todo.push (s);
      if $\varepsilon \models s$ **then**
        $F \leftarrow F \cup \{s\}$;
    $\delta \leftarrow \delta \cup \{f \xrightarrow{p} s\}$;
return $\langle Q, \delta, \varphi, F \rangle$;

Initial state is the input formula φ

Loop over LF pairs

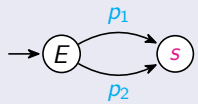
States are labeled by SEREs

Identify accepting states

Two Simplifications on Linear Forms

Unique Suffix

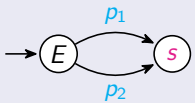
$$\mathsf{LF}(E) = \{\ldots, (p_1, s), (p_2, s), \ldots\}$$



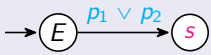
Two Simplifications on Linear Forms

Unique Suffix

$\mathrm{LF}(E) = \{\ldots, (p_1, s), (p_2, s), \ldots\}$



$\mathrm{US}(\mathrm{LF}(E)) = \{\ldots, (p_1 \vee p_2, s), \ldots\}$

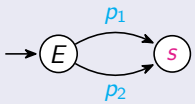


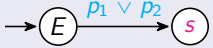
Two Simplifications on Linear Forms

Unique Suffix

$\mathrm{LF}(E) = \{\ldots, (p_1, s), (p_2, s), \ldots\}$

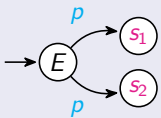
$\mathrm{US}(\mathrm{LF}(E)) = \{\ldots, (p_1 \vee p_2, s), \ldots\}$





Unique Prefix

$\mathrm{LF}(E) = \{\ldots, (p, s_1), (p, s_2), \ldots\}$

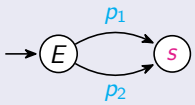


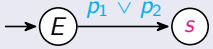
Not a determinization because p_is are Boolean formulas.

Two Simplifications on Linear Forms

Unique Suffix

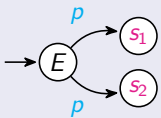
$\mathrm{LF}(E) = \{\ldots, (p_1, s), (p_2, s), \ldots\}$ $\quad$ $\mathrm{US}(\mathrm{LF}(E)) = \{\ldots, (p_1 \vee p_2, s), \ldots\}$

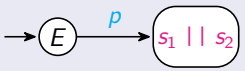




Unique Prefix

$\mathrm{LF}(E) = \{\ldots, (p, s_1), (p, s_2), \ldots\}$ $\quad$ $\mathrm{UP}(\mathrm{LF}(E)) = \{\ldots, (p, s_1 \mid\mid s_2), \ldots\}$





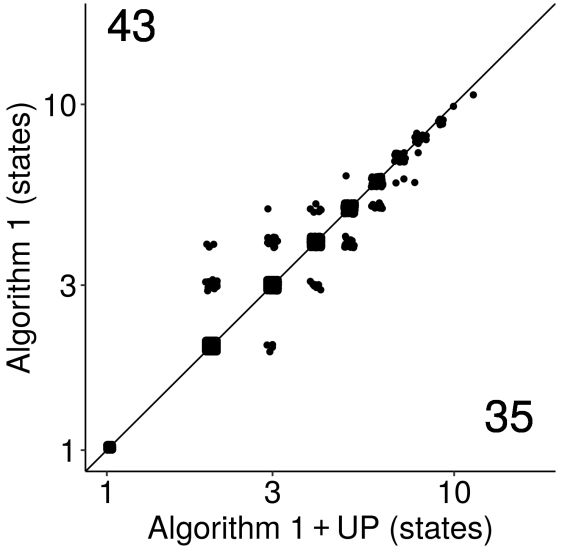
Not a determinization because p_is are Boolean formulas.

Benchmarking UP Effects

Benchmark dataset

12500 SEREs, randomly generated with Spot www. We enforced some heterogeneity in the dataset.

Some jitter was added to the plot



Linear Form Equality

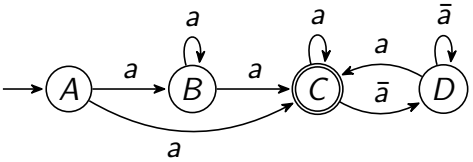
Property

$$\mathsf{LF}(E) = \mathsf{LF}(F) \implies \mathscr{L}(E)\backslash\{\varepsilon\} = \mathscr{L}(F)\backslash\{\varepsilon\}$$

$$\mathsf{LF}(\mathtt{a[*]}) = \mathsf{LF}(\mathtt{a\ ;\ a[*]}) = \{(\mathtt{a}, \mathtt{a[*]})\}$$

Improving the Translation

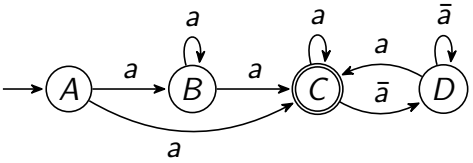
Algo 1:



$$\mathrm{LF}(A) = \mathrm{LF}(B)$$
$$\mathrm{LF}(C) = \mathrm{LF}(D)$$

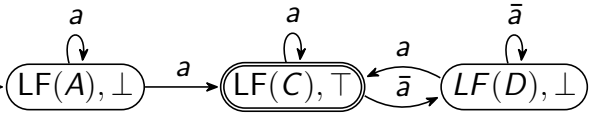
Improving the Translation

Algo 1:

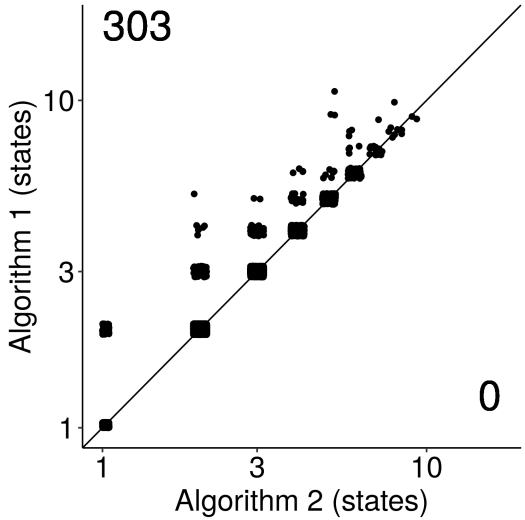


$$\mathrm{LF}(A) = \mathrm{LF}(B)$$
$$\mathrm{LF}(C) = \mathrm{LF}(D)$$

Algo 2: labeling with linear forms

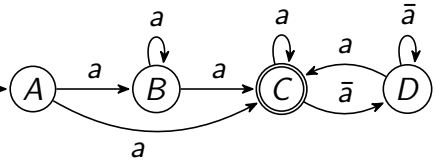


Algorithm 2 is an Improvement



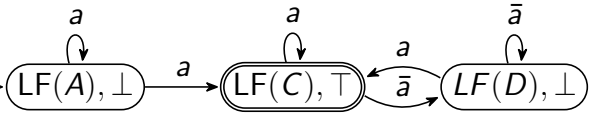
Improving the Translation

Algo 1:



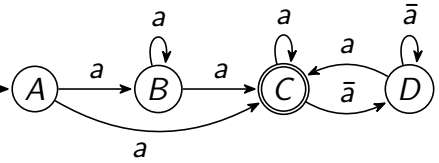
$$\mathrm{LF}(A) = \mathrm{LF}(B)$$
$$\mathrm{LF}(C) = \mathrm{LF}(D)$$

Algo 2: labeling with linear forms



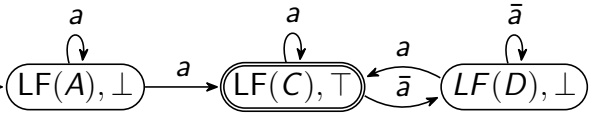
Improving the Translation

Algo 1:

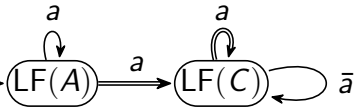


$$\mathrm{LF}(A) = \mathrm{LF}(B)$$
$$\mathrm{LF}(C) = \mathrm{LF}(D)$$

Algo 2: labeling with linear forms



Algo 3: using transition-based acceptance



Algorithm 1 vs Algorithm 3

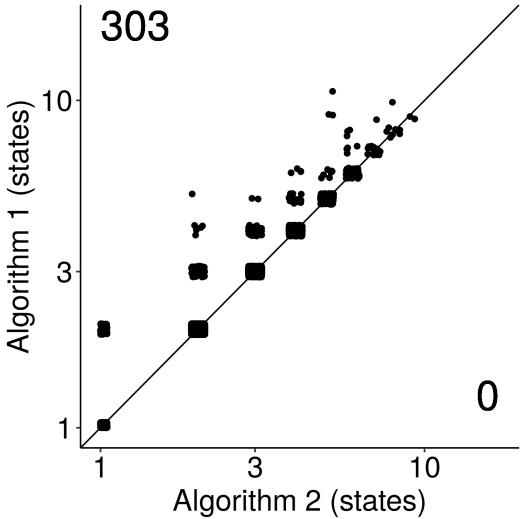
input : A SERE φ
output: An NFA $\mathcal{A}$ such that $\mathscr{L}(\mathcal{A}) = \mathscr{L}(\varphi)$

$Q, \delta, F \leftarrow \{\varphi\}, \varnothing, \varnothing$;
todo.push(φ);
while todo $\neq \varnothing$ **do**
  $f \leftarrow$ todo.pop();
  foreach $(p, s) \in \mathrm{LF}(f)$ **do**
    if $s \notin Q$ **then**
      $Q \leftarrow Q \cup \{s\}$;
      todo.push (s);
      if $\varepsilon \models s$ **then**
        $F \leftarrow F \cup \{s\}$;
    $\delta \leftarrow \delta \cup \{f \xrightarrow{p} s\}$;
return $\langle Q, \delta, \varphi, F \rangle$;

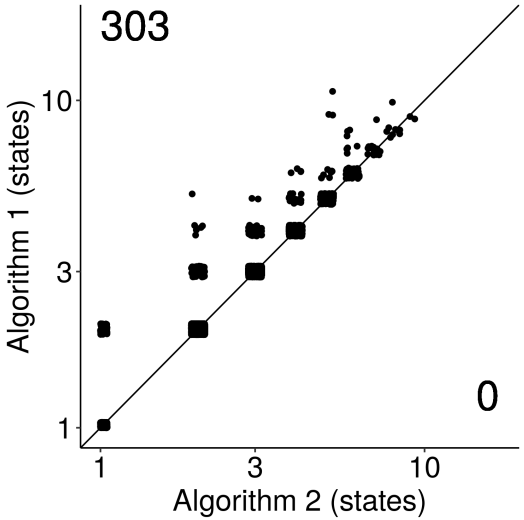
input : A SERE φ
output: A tNFA $\mathcal{A}$ such that $\mathscr{L}(\mathcal{A}) = \mathscr{L}(\varphi)$

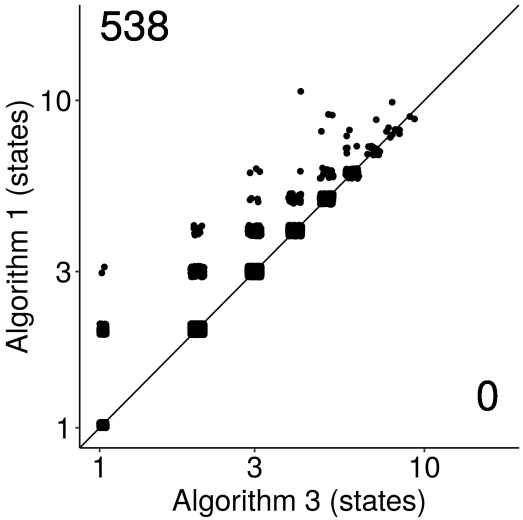
$L_\iota, b_\iota \leftarrow \mathrm{LF}(\phi), [\lambda(\varphi) = \varepsilon]$;
$Q, \delta \leftarrow \{L_\iota\}, \varnothing$;
todo.push(L_ι);
while todo $\neq \varnothing$ **do**
  $L \leftarrow$ todo.pop();
  foreach $(p, s) \in L$ **do**
    $L', b' \leftarrow \mathrm{LF}(s), [\lambda(s) = \varepsilon]$;
    if $L' \notin Q$ **then**
      $Q \leftarrow Q \cup \{L'\}$;
      todo.push (L');
    $\delta \leftarrow \delta \cup \{L \xrightarrow{p, b'} L'\}$;
return $\langle Q, \delta, L_\iota, b_\iota \rangle$;

Algorithm 3 is Better

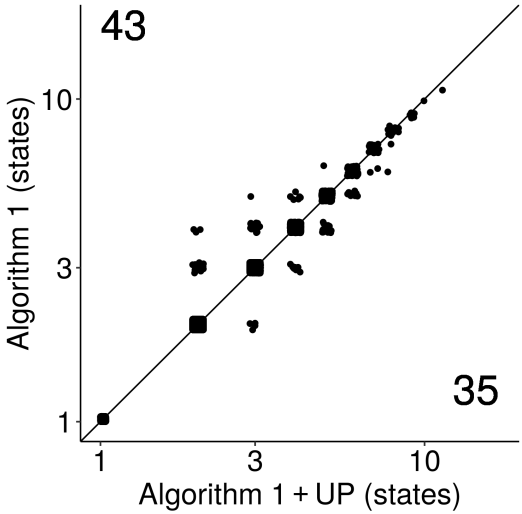


Algorithm 3 is Better

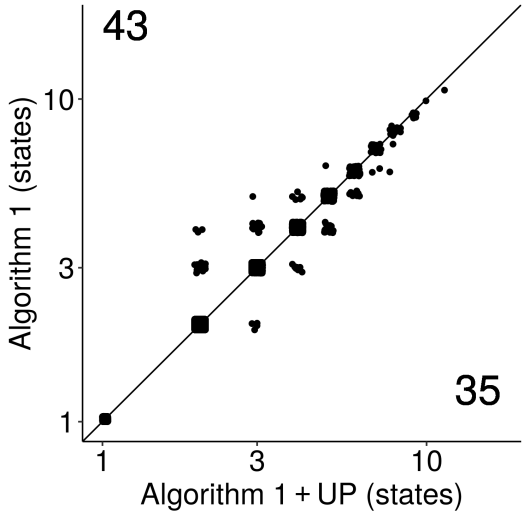


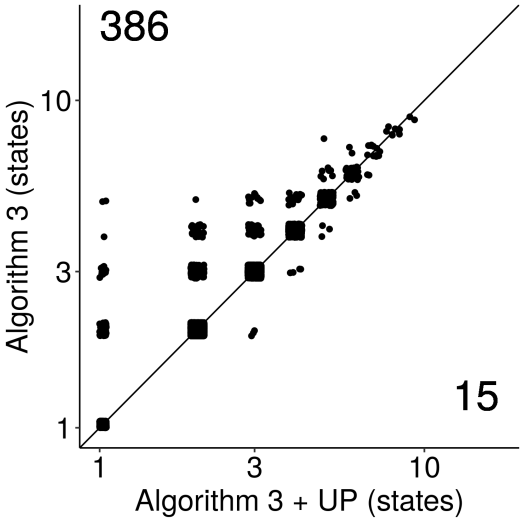


UP Works Better with Algorithm 3



UP Works Better with Algorithm 3





Wrapping up

Contributions

- Extended Antimirov's linear forms
  - Support alphabet $\Sigma = 2^{AP}$
  - Support SERE operators
- Evaluation of UP / US simplifications' performance
- Labeling of automaton states with linear forms
- Translation using transition-based acceptance

Wrapping up

Contributions

- Extended Antimirov's linear forms
  - Support alphabet $\Sigma = 2^{AP}$
  - Support SERE operators
- Evaluation of UP / US simplifications' performance
- Labeling of automaton states with linear forms
- Translation using transition-based acceptance

Future work

- Build on this to translate PSL / SVA to Büchi automata
- Evaluate performance of translation in a model checking context.