Translation of Semi-Extended Regular Expressions using Linear Forms

Antoine Martin[a,b], Etienne Renault[c], Alexandre Duret-Lutz[b]

[a] *LIPADE, Université Paris Cité, 45 rue des Saints Pères, Paris, France*
[b] *LRE, EPITA, 14-16 rue Voltaire, Le Kremlin-Bicêtre, France*
[c] *SiPearl, 44 rue Jean Mermoz, Maisons-Laffitte, France*

Abstract

We generalize Antimirov's notion of *linear form* of a regular expression, to the Semi-Extended Regular Expressions typically used in the Property Specification Language or SystemVerilog Assertions. Doing so requires extending the construction to handle more operators, and dealing with expressions over alphabets $\Sigma = 2^{AP}$ of valuations of atomic propositions. Using linear forms to construct automata labeled by Boolean expressions suggests heuristics that we evaluate. Finally, we study a variant of this translation to automata with accepting transitions: this construction is more natural and provides smaller automata.

Keywords: Regular expressions, Automata, PSL, SVA, Derivative

1. Introduction

In this paper we discuss and compare techniques for translating (extended) regular expressions over alphabets $\Sigma = 2^{AP}$ where letters describe valuations of a set of atomic propositions AP. Such alphabets are typically used in formal methods such as *model checking* [3], *runtime verification* [4] or *synthesis* [17]. In our case, we are interested in supporting the regular expression operators of the PSL and SVA standards.

Property Specification Language (PSL) [16] and SystemVerilog Assertions (SVA) [1] are two industrial formal verification languages used in the field of hardware design and verification. These two languages include features for describing linear-time temporal properties or reasoning with clocks, but we restrict ourselves to the (semi-extended) regular expression properties. Both offer a nearly identical set of operators, albeit with different syntaxes.

The PSL expression "`btn : ((red[=2] && opn[->]) || rst[->])`", for instance, matches any sequence that starts with the `btn` signal on, and in which either the `red` signal is on exactly twice in the interval it takes for the signal `opn` to turn on, or in which the `rst` signal turns on. In SVA this expression becomes "`btn ##0 ((red[=2] intersect opn[->1]) or rst[->1])`".

Other logics such as Linear Dynamic Logic (LDL) [19] or Propositional Dynamic Logic (PDL) [18] also use regular expressions over atomic propositions, so our work applies to them too. Unlike SVA or PSL, however, these logics are usually defined only with classical regular operators plus a $\varphi?$ operator that is absent from PSL and SVA, and that we do not consider.

This paper is an extension of our CIAA'24 publication [24]. This version includes proofs for all theorems, details our determinization algorithm, provides additional discussions about the experiments (redone with some fixes), and in particular, it sheds some light on the interactions between the different simplifications and the translation algorithms proposed.

2. Definitions

In the entire paper, we assume an alphabet $\Sigma = 2^{AP}$ where letters describe valuations of a set of *atomic propositions* AP. For instance, if $AP = \{a, b\}$ then we denote the valuations as $\Sigma = \{\bar{a}\bar{b}, \bar{a}b, a\bar{b}, ab\}$. We use $\Sigma^\star$ to denote the set of finite sequences of valuations. For a sequence $\sigma \in \Sigma^\star$, we denote $|\sigma|$ its length, and we write $\sigma(i)$ for the letter at position $i \in \{0, 1, \ldots, |\sigma|-1\}$ in σ. Using " ;" as concatenation operator, we write $\sigma = \sigma(0)\,;\sigma(1)\,;\ldots;\sigma(|\sigma|-1)$. Finally, for two integers i, j such that $0 \leq i \leq j \leq |\sigma|$, we write $\sigma^{i..j}$ the (possibly empty if $i = j$) subsequence $\sigma(i)\ ;\ \sigma(i+1)\ ;\ \ldots\ ;\ \sigma(j-1)$. For convenience, we write $\sigma^{i..}$ instead of $\sigma^{i..|\sigma|}$ and $\sigma^{..j}$ instead of $\sigma^{0..j}$.

Definition 1 (Boolean expression)**.** *Any Boolean expression b is built using the following grammar, where $a \in AP$ can be any atomic proposition.*

$$b ::= \bot \mid \top \mid a \mid (b \vee b) \mid (b \wedge b) \mid \neg b$$

For convenience, we omit unnecessary parentheses, and use operators $\rightarrow$ and $\leftrightarrow$ as syntactic sugar with their obvious definitions.

Boolean expression are interpreted over a valuation $v \in \Sigma$ in the obvious way. We write $v \models b$ when the valuation v satisfies b, and $b \equiv b'$ when two Boolean expressions b and b' are satisfied by the same valuations.[1]

[1] Testing $b \equiv b'$ is straightforward if b and b' are represented with BDDs [8].

We use $\mathbb{B} = \{\bot, \top\}$ to denote the set of Boolean values, and $\mathbb{B}(AP)$ to denote the set of Boolean expressions over AP.

Definition 2 (SERE)**.** *A Semi-Extended Regular Expression (SERE) r is built using the following grammar:*

$$r ::= b \mid \varepsilon \mid (r\,;r) \mid (r:r) \mid r^\star \mid (r \vee r) \mid (r \wedge r) \mid \mathsf{fm}(r)$$

The symbol "ε" is called the empty word*. Operators "$\vee$" (choice), ";" (concatenation) and "*" (Kleene star) are traditional regular operators. SERE extends those with "$\wedge$" (intersection) ":" (fusion), and "*fm*" (SVA's* first-match*). In practice, we omit parentheses when they are not necessary.*

The set of all SEREs is written SERE*.*

SEREs are interpreted over a finite sequence $\sigma \in \Sigma^\star$ of valuations defined inductively as follows:

$$\begin{array}{rcl}
\sigma \models b & \textit{iff} & |\sigma| = 1 \wedge \sigma(0) \models b \\
\sigma \models \varepsilon & \textit{iff} & |\sigma| = 0 \\
\sigma \models (r_1\,;r_2) & \textit{iff} & \exists i \geq 0,\ \sigma^{..i} \models r_1 \wedge \sigma^{i..} \models r_2 \\
\sigma \models (r_1 : r_2) & \textit{iff} & \exists i \geq 0,\ \sigma^{..i+1} \models r_1 \wedge \sigma^{i..} \models r_2 \\
\sigma \models r^\star & \textit{iff} & \textit{either } |\sigma| = 0 \textit{ or } \exists i > 0,\ \sigma^{..i} \models r \wedge \sigma^{i..} \models r^\star \\
\sigma \models (r_1 \vee r_2) & \textit{iff} & \sigma \models r_1 \vee \sigma \models r_2 \\
\sigma \models (r_1 \wedge r_2) & \textit{iff} & \sigma \models r_1 \wedge \sigma \models r_2 \\
\sigma \models \mathsf{fm}(r) & \textit{iff} & (\sigma \models r) \wedge (\forall i < |\sigma|,\ \sigma^{..i} \not\models r)
\end{array}$$

The language *of a SERE r is the set $\mathscr{L}(r) = \{\sigma \in \Sigma^\star \mid \sigma \models r\}$ of all sequences satisfying r (or "matched" by r).*

In the above definition, regular expressions have been extended with three operators: the conjunction "$\wedge$" has obvious meaning, the fusion operator "$:$" ensures that the last letter matching the left operand is also the first letter matching the right operand (this implies that a fusion can never match the empty word), and finally fm, the first-match operator of SVA, retains only the shortest possible match for a SERE r.

The PSL and SVA specifications define other SERE operators (such as `[=`n`]` or `[->`n`]`) that can be seen as syntactic sugar on the above. In our syntax, the expression from the introduction becomes

$$\varphi = btn{:}\Big(\big(((\neg red)^\star;red;(\neg red)^\star;red;(\neg red)^\star) \wedge ((\neg opn)^\star;opn)\big) \vee \big((\neg rst)^\star;rst\big)\Big)$$

Definition 3 (Syntactic equivalence)**.** *Given two SEREs r_1 and r_2, we say that they are syntactically equivalent, denoted $r_1 \stackrel{\circ}{=} r_2$, if one can be rewritten into the other using the following so called ACI-rules (associativity, commutativity, and idempotence) and a few others:*

$$(r_1 \odot r_2) \odot r_3 \stackrel{\circ}{=} r_1 \odot (r_2 \odot r_3) \stackrel{\circ}{=} r_1 \odot r_2 \odot r_3 \quad \text{for } \odot \in \{;, :, \vee, \wedge\} \tag{A}$$

$$r_1 \odot r_2 \stackrel{\circ}{=} r_2 \odot r_1 \quad \text{for } \odot \in \{\vee, \wedge\} \tag{C}$$

$$r_1 \odot r_1 \stackrel{\circ}{=} r_1 \quad \text{for } \odot \in \{\vee, \wedge\} \tag{I1}$$

$$r^{\star\star} \stackrel{\circ}{=} r^{\star} \qquad \mathsf{fm}(\mathsf{fm}(r)) \stackrel{\circ}{=} \mathsf{fm}(r) \tag{I2}$$

$$r \vee \bot \stackrel{\circ}{=} r \qquad r \wedge \top \stackrel{\circ}{=} r \qquad r\,;\varepsilon \stackrel{\circ}{=} \varepsilon\,;r \stackrel{\circ}{=} r \tag{Z}$$

$$r \wedge \bot \stackrel{\circ}{=} \bot \qquad r\,;\bot \stackrel{\circ}{=} \bot\,;r \stackrel{\circ}{=} \bot \tag{U1}$$

$$r \vee \top \stackrel{\circ}{=} \top \qquad r : \bot \stackrel{\circ}{=} \bot : r \stackrel{\circ}{=} \bot \tag{U2}$$

$$r : \varepsilon \stackrel{\circ}{=} \varepsilon : r \stackrel{\circ}{=} \bot \tag{U3}$$

$$\mathsf{fm}(r) \stackrel{\circ}{=} \varepsilon \text{ if } \varepsilon \models r \tag{F}$$

Proposition 1. *Two syntactically equivalent SEREs have the same language. I.e., $(r_1 \stackrel{\circ}{=} r_2) \implies (\mathscr{L}(r_1) = \mathscr{L}(r_2))$.*

Proof. The proof follows from the fact that rewritings (A)–(F) are language-preserving. As most of these rules are either well known or obvious, we only prove those involving fm. For I2 we have:

$$\begin{aligned}
\sigma \models \mathsf{fm}(\mathsf{fm}(r)) &\iff (\sigma \models \mathsf{fm}(r)) \wedge (\forall i < |\sigma|,\, \sigma^{\cdot\cdot i} \not\models \mathsf{fm}(r)) \\
&\iff (\sigma \models r) \wedge (\forall i < |\sigma|,\, \sigma^{\cdot\cdot i} \not\models r) \wedge (\forall i < |\sigma|,\, \sigma^{\cdot\cdot i} \not\models \mathsf{fm}(r)) \\
&\iff (\sigma \models r) \wedge (\forall i < |\sigma|,\, \neg(\sigma^{\cdot\cdot i} \models r \vee \underbrace{\sigma^{\cdot\cdot i} \models \mathsf{fm}(r)}_{\text{implies } \sigma^{\cdot\cdot i} \models r \text{ by definition}})) \\
&\iff (\sigma \models r) \wedge (\forall i < |\sigma|,\, \sigma^{\cdot\cdot i} \not\models r) \quad \iff \quad \sigma \models \mathsf{fm}(r)
\end{aligned}$$

Therefore, $\mathscr{L}(\mathsf{fm}(\mathsf{fm}(r))) = \mathscr{L}(\mathsf{fm}(r))$. Similarly, for rule (F), we have

$$\begin{aligned}
(\sigma \models \mathsf{fm}(r)) \wedge (\varepsilon \models r) &\iff (\sigma \models r) \wedge \underbrace{(\forall i < |\sigma|,\, \sigma^{\cdot\cdot i} \not\models r) \wedge (\varepsilon \models r)}_{\text{implies } |\sigma|=0 \text{ otherwise } \sigma^{\cdot\cdot 0}=\varepsilon \not\models r \text{ conflicts with } \varepsilon \models r} \\
&\iff |\sigma| = 0 \iff \sigma \models \varepsilon
\end{aligned}$$

Therefore when $\varepsilon \models r$, $\mathscr{L}(\mathsf{fm}(r)) = \mathscr{L}(\varepsilon)$. □

From an implementation standpoint, it is straightforward to rewrite any SERE r into a unique representative of its equivalence class $[r]_{\stackrel{\circ}{=}} = \{s \in$

$\mathsf{SERE} \mid r \stackrel{\circ}{=} s\}$. This can be achieved by applying the above rewriting rules during the construction of the syntax tree of r. In particular rule (A) can be implemented by considering these operators as n-ary (rather than binary), and then rules (C) and (I1) can be implemented by sorting operands and removing duplicates. Rule (F) can be implemented at construction as well, by deciding $\varepsilon \models r$ using Proposition 2 below. From now on, we assume that these rules are automatically applied at construction to any SERE we handle.

Definition 4 (Constant Term)**.** *The* constant term *of an expression r, denoted $\lambda(r) \in \{\bot, \varepsilon\}$ is a SERE defined inductively as follows for any Boolean formula b and any SEREs r_1, r_2.*

$$\begin{aligned} \lambda(b) &= \bot & \lambda(r_1 \vee r_2) &= \lambda(r_1) \vee \lambda(r_2) \\ \lambda(\varepsilon) &= \varepsilon & \lambda(r_1 \wedge r_2) &= \lambda(r_1) \wedge \lambda(r_2) \\ \lambda(r_1 : r_2) &= \bot & \lambda(r_1 \,;\, r_2) &= \lambda(r_1) \,;\, \lambda(r_2) \\ \lambda(r_1^\star) &= \varepsilon & \lambda(\mathsf{fm}(r_1)) &= \lambda(r_1) \end{aligned}$$

The above rules for $\vee$, $\wedge$, and ; assume that the simplifications (I1), (Z), and (U1) are used, so that the result is always an element of $\{\bot, \varepsilon\}$.

Proposition 2. *With the above notation, $\lambda(r) = \varepsilon$ iff $\varepsilon \models r$.*

Definition 5 (NFA)**.** *A* nondeterministic finite automaton *is a tuple $\mathcal{A} = \langle Q, \delta, \iota, F\rangle$ where Q is a finite set of states, $\delta \subseteq Q \times \mathbb{B}(AP) \times Q$ is the transition relation, $\iota \in Q$ is the initial state, and $F \subseteq Q$ is the set of final states.*

We write $s \xrightarrow{f} d$ when $(s, f, d) \in \delta$.

A sequence of valuations $\sigma \in \Sigma^n$ of size n is accepted *by $\mathcal{A}$ if either $n = 0$ and $\iota \in F$, or $n > 0$ and there exists a sequence of transitions $\rho = s_0 \xrightarrow{f_0} s_1 \xrightarrow{f_1} \cdots \xrightarrow{f_{n-1}} s_n$ such that $s_0 = \iota$, $s_n \in F$, and for each i, $\sigma(i) \models f_i$.*

The language *of $\mathcal{A}$, denoted $\mathscr{L}(\mathcal{A})$ is the set of words accepted by $\mathcal{A}$.*

A Deterministic Finite Automaton *(DFA) is an NFA where transitions leaving each state have mutually exclusive Boolean expressions. Formally, an automaton is a DFA if for any two different transitions $s \xrightarrow{f} d$ and $s \xrightarrow{f'} d'$ with the same origin, we have $f \wedge f' \equiv \bot$.*

Such automata are sometimes called symbolic finite automata [12], however in our case the alphabet is always finite, so they can be handled in a usual way.

3. Building Automata using Linear Forms

In 1964, Brzozowski [9] introduced the notion of *derivative* of a regular expression, allowing the construction of an equivalent deterministic finite automaton. This work was extended in 1995 by Antimirov [2], with a notion of *partial derivatives* allowing the construction of a non-deterministic finite automaton. More importantly, Antimirov introduced the concept of *linear form* of a regular expression as a more efficient way to compute the set of *partial derivatives* without having to iterate over the entire alphabet. Similar approaches, that construct all successors of an expression at once, have been devised for other formalisms such as KAT, which combines the algebra of regular expressions with Boolean algebra [6], its synchronous variant, SKAT [7], and temporal logic over finite-traces [15].

An extension of *partial derivatives* was proposed by Caron et al. [10] to handle intersection and complement. Here, we adapt the concept of *linear form*, to SERE with an alphabet over 2^{AP} and their specific operators. In particular, the fact that our alphabet is exponential in the number of atomic propositions makes *linear forms* much more attractive than *partial derivatives*, because using the latter to build an automaton requires iterating over exponentially many letters.

3.1. Linear Forms

Definition 6 (Linear Form)**.** *A* linear form *for a SERE r is a finite set of pairs $\{(p_1, s_1), (p_2, s_2), \ldots\} \subseteq 2^{\mathbb{B}(AP)\times\mathsf{SERE}}$ where $p_i \in \mathbb{B}(AP)$, $p_i \not\equiv \bot$, $s_i \in \mathsf{SERE}$ and $s_i \not\stackrel{\circ}{=} \bot$, such that $\bigcup_i \mathscr{L}(p_i\,;s_i) = \mathscr{L}(r) \setminus \{\varepsilon\}$.*

This definition differs from that of Antimirov [2] in that p_i is a satisfiable Boolean expression rather than a letter, and in that we explicitly forbid $s_i \stackrel{\circ}{=} \bot$. To simplify the upcoming notations we assume that any pair (p_i, s_i) in a linear form we construct is implicitly ignored when $p_i \equiv \bot$; for instance we shall write $\{\ldots, (p_i \wedge \neg p_j, s_i), \ldots\}$ with the implicit assumption that the pair $(p_i \wedge \neg p_j, s_i)$ must be omitted when $p_i \wedge \neg p_j$ is not satisfiable.

As we saw in Proposition 2, an empty sequence may only be matched by a SERE r if $\lambda(r) = \varepsilon$. If a non-empty sequence σ is matched by a SERE r, then Definition 6 implies that a linear form for r will have at least one pair (p_i, s_i) such that $\sigma(0) \models p_i$ and $\sigma^{1\cdot\cdot} \models s_i$. As a mental model for a linear form, it is useful to interpret it as the partial automaton shown in Figure 1: the p_is are Boolean formulas evaluated against $\sigma(0)$, and the s_is tell what SERE should be checked against the suffix $\sigma^{1\cdot\cdot}$. The constraints,

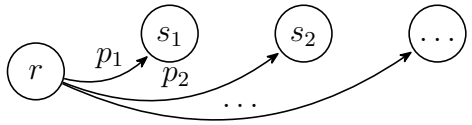

Figure 1: Automaton view of a linear form $\{(p_1, s_1), (p_2, s_2), \ldots\}$ for a SERE r.

in Definition 6, that $p_i \not\equiv \bot$, and $s_i \not\equiv \bot$, prevent the creation of paths that will not recognize any word.

A SERE may have multiple linear forms; some will be called *deterministic*.

Definition 7 (Deterministic Linear Form)**.** *A linear form* $\{(p_1, s_1), (p_2, s_2), \ldots\}$ *is* deterministic *if for any* $i \neq j$, $p_i \wedge p_j \equiv \bot$.

Here is a linear form for the formula φ of Section 2.

$$\begin{aligned} L_1 = \big\{ &\big(btn\wedge\neg red\wedge\neg opn, ((\neg red)^\star; red; (\neg red)^\star; red; (\neg red)^\star) \wedge ((\neg opn)^\star; opn)\big),\\ &\big(btn\wedge red\wedge\neg opn, \quad ((\neg red)^\star; red; (\neg red)^\star) \wedge ((\neg opn)^\star; opn)\big),\\ &\big(btn\wedge\neg rst, \qquad (\neg rst)^\star; rst\big),\\ &\big(btn\wedge rst, \qquad \varepsilon\big)\big\} \end{aligned}$$

L_1 is not deterministic because, for instance, $btn\wedge red\wedge\neg opn$ can hold together with $btn\wedge rst$. Here is a deterministic linear form for φ:

$$\begin{aligned} L_2 = \big\{ &\big(btn\wedge\neg red\wedge\neg opn\wedge\neg rst,\\ &\quad (((\neg red)^\star; red; (\neg red)^\star; red; (\neg red)^\star) \wedge ((\neg opn)^\star; opn)) \vee ((\neg rst)^\star; rst)\big),\\ &\big(btn\wedge\neg red\wedge\neg opn\wedge rst,\\ &\quad (((\neg red)^\star; red; (\neg red)^\star; red; (\neg red)^\star) \wedge ((\neg opn)^\star; opn)) \vee \varepsilon\big),\\ &\big(btn\wedge red\wedge\neg opn\wedge\neg rst,\\ &\quad (((\neg red)^\star; red; (\neg red)^\star) \wedge ((\neg opn)^\star; opn)) \vee ((\neg rst)^\star; rst)\big),\\ &\big(btn\wedge red\wedge\neg opn\wedge rst, \quad (((\neg red)^\star; red; (\neg red)^\star) \wedge ((\neg opn)^\star; opn)) \vee \varepsilon\big)\big\} \end{aligned}$$

Property 1. *Any linear form* $\{(p_1, s_1), (p_2, s_2), \ldots\}$ *for an expression* r *can be converted into a deterministic linear form for* r.

The idea is that if $p_1 \wedge p_2 \not\equiv \bot$, then $\{(p_1 \wedge \neg p_2, s_1), (\neg p_1 \wedge p_2, s_2), (p_1 \wedge p_2, s_1 \vee s_2), \ldots\}$ is also a linear form for r

input : A linear form L
output: An equivalent deterministic linear form

$L' \leftarrow \varnothing$
foreach $(p_2, s_2) \in L$ **do**
  $L'' \leftarrow \varnothing$
  foreach $(p_1, s_1) \in L'$ **do**
    if $p_1 \wedge p_2 \neq \bot$ **then**
      $L'' \leftarrow L'' \cup \{(p_1 \wedge \neg p_2, s_1), (p_1 \wedge p_2, s_1 \vee s_2)\}$
      $p_2 \leftarrow p_2 \wedge \neg p_1$
  $L' \leftarrow L'' \cup \{(p_2, s_2)\}$
return L'

Algorithm 1: Determinization of a linear form.

Algorithm 1 shows an implementation of this idea by adding the pairs $(p_2, s_2) \in L$ one by one to L' and splitting any pair $(p_1, s_1) \in L'$ that intersect the pair (p_2, s_2).

From now on, we assume the existence of a function det that determinizes a linear form. It could be Algorithm 1 or any other.

3.2. Linearization of SEREs

We now discuss how to convert a SERE into a linear form. For now on, we assume that equations (A)–(F) are always applied, i.e., that we are only working with unique representatives of each equivalence classes of $\mathrel{\mathring{=}}$, as discussed in Section 2.

To simplify the notations, we extend the concatenation and fusion operators to linear forms: given a linear form $L = \{(p_1, s_1), (p_2, s_2), \ldots, (p_n, s_n)\}$, an operator $\odot \in \{\,;, :\}$, and a SERE r, we write $L \odot r$ instead of $\{(p, s \odot r) \mid (p, s) \in L, s \odot r \neq \bot\}$. The notation works similarly for $r \odot L$.

The following LF function turns a SERE into a linear form. It mostly extends the "*lf*" function of Antimirov [2, eq. (45)–(51)] to deal with SERE operators and Boolean formulas.

Definition 8 (Linearization of a SERE)**.** *Assuming that b is a Boolean formula, and that r, r_1, and r_2 are SEREs, we define* $\mathsf{LF} : \mathsf{SERE} \to 2^{\mathbb{B}(AP) \times \mathsf{SERE}}$

as follows.

$$
\begin{aligned}
\mathsf{LF}(\bot) &= \varnothing \\
\mathsf{LF}(\varepsilon) &= \varnothing \\
\mathsf{LF}(b) &= \{(b, \varepsilon)\} \\
\mathsf{LF}(r_1 \vee r_2) &= \mathsf{LF}(r_1) \cup \mathsf{LF}(r_2) \\
\mathsf{LF}(r^\star) &= \mathsf{LF}(r)\,;r^\star \\
\mathsf{LF}(r_1\,;r_2) &= (\mathsf{LF}(r_1)\,;r_2) \cup (\lambda(r_1)\,;\mathsf{LF}(r_2)) \\
\mathsf{LF}(r_1 : r_2) &= (\mathsf{LF}(r_1) : r_2) \cup \left\{ (p_i \wedge p_j, s_j) \;\middle|\; \begin{array}{l} (p_i, s_i) \in \mathsf{LF}(r_1),\ \lambda(s_i) = \varepsilon, \\ (p_j, s_j) \in \mathsf{LF}(r_2) \end{array} \right\} \\
\mathsf{LF}(r_1 \wedge r_2) &= \{(p_i \wedge p_j, s_i \wedge s_j) \mid (p_i, s_i) \in \mathsf{LF}(r_1),\ (p_j, s_j) \in \mathsf{LF}(r_2)\} \\
\mathsf{LF}(\mathsf{fm}(r)) &= \{(p_i, \mathsf{fm}(s_i)) \mid (p_i, s_i) \in \mathsf{det}(\mathsf{LF}(r))\}
\end{aligned}
$$

As noted below Definition 6, we assume that when one of these equations generates a pair (p_i, s_i) with $p_i \equiv \bot$, it is implicitly removed. Since we assume that rules (A)–(F) are always applied, these equations cannot produce a pair (p_i, s_i) such that $s_i \stackrel{\circ}{=} \bot$, but it could nonetheless be the case that $\mathscr{L}(s_i) = \varnothing$ (for instance if $s_i = a \wedge \neg a$).

The above definition is ambiguous for $\mathsf{LF}(a_1 \vee a_2)$ where $a_1, a_2 \in AP$. It can generate $\{(a_1, \varepsilon), (a_2, \varepsilon)\}$ or $\{(a_1 \vee a_2, \varepsilon)\}$ depending on whether $a_1 \vee a_2$ is considered as a Boolean formula or as a disjunction of two SEREs. Both choices are valid, but our implementation will always prioritize the rule $\mathsf{LF}(b) = \{(b, \varepsilon)\}$ when it applies, to keep the linear form short.

To understand the definition for $\mathsf{LF}(\mathsf{fm}(r))$, it may be useful to give an intuition of how $\mathsf{fm}(r)$ works. The SERE $\mathsf{fm}(r)$ may match σ if only if σ is the shortest prefix of σ matching r. An easy way to construct an automaton for $\mathsf{fm}(r)$ is therefore to build a DFA for r, and then remove the outgoing edges of all accepting states of that DFA. This is actually what the above definition achieves. The use of $\mathsf{det}(\mathsf{LF}(r))$ is making sure that the linear form is deterministic, and the use of $\mathsf{fm}(s_i)$ serves two purposes: (1) it ensures that upcoming choices will still be deterministic, and (2) more importantly, by applying rule (F), it cuts the successors of s_i when $\varepsilon \models s_i$. For instance, $\mathsf{LF}(\mathsf{fm}(a\,;a^\star)) = \{(a, \varepsilon)\}$ because $\mathsf{fm}(a^\star)$ gets reduced to ε by rule (F).

To use LF in an algorithm for building an automaton that recognizes $\mathscr{L}(r)$, we need two theorems. First, $\mathsf{LF}(r)$ should be a linear form, i.e., it should preserve the language of r, except for the empty word (Theorem 1

below). Then, the number of new expressions that can be created by applying LF recursively has to be finite (this will be Theorem 2 later).

Theorem 1. $\mathsf{LF}(r) = \{(p_1, s_1), (p_2, s_2), \ldots\}$ *is a linear form for* $r \in \mathsf{SERE}$.

Proof. Let $\mathscr{L}(\mathsf{LF}(r))$ designate $\bigcup_i \mathscr{L}(p_i\,;s_i)$. Using Definitions 2 and 8, we can show by induction on the structure of r that $\mathscr{L}(\mathsf{LF}(r)) = \mathscr{L}(r)\setminus\{\varepsilon\}$. For the regular operators, it was already established by Antimirov [2, Prop. 2.5].

We show here the case of $r_1 : r_2$. For $m \in \{1, 2\}$ assume that $\mathsf{LF}(r_m) = \{(p_1^m, s_1^m), (p_2^m, s_2^m), \ldots\}$ is a linear form for r_m, i.e., that $\bigcup_i \mathscr{L}(p_i^m\,;s_i^m) = \mathscr{L}(r_m) \setminus \{\varepsilon\}$. We would like to prove that $\mathscr{L}(\mathsf{LF}(r_1 : r_2)) = \mathscr{L}(r_1 : r_2) \setminus \{\varepsilon\}$.

Since $\mathscr{L}(r_1{:}r_2)$ may not contain ε by definition, we just have to prove that for any sequence $\sigma \in \Sigma^*$ we have $\sigma \models \mathscr{L}(\mathsf{LF}(r_1 : r_2)) \iff \sigma \models \mathscr{L}(r_1 : r_2)$.

($\Longleftarrow$) Consider σ in $\mathscr{L}(r_1 : r_2)$. By Definition 2 there exists $k \geq 0$ such that $\sigma^{\cdot\cdot k+1} \models r_1$ and $\sigma^{k\cdot\cdot} \models r_2$.

If $k = 0$, $\sigma^{\cdot\cdot 1} \models r_1$ implies that there exists a pair (p_i^1, s_i^1) in $\mathsf{LF}(r_1)$ such that $\sigma(0) \models p_i^1$ and $\lambda(s_i^1) = \varepsilon$. Furthermore since $\sigma \models r_2$, there exists a pair (p_j^2, s_j^2) in $\mathsf{LF}(r_2)$ such that $\sigma(0) \models p_j^2$. We can see in Definition 8 that the pair $(p_i^1 \wedge p_j^2, s_j^2)$ exists in $\mathsf{LF}(r_1 : r_2)$, therefore $\sigma \in \mathscr{L}(\mathsf{LF}(r_1 : r_2))$.

If $k > 0$, $\sigma^{\cdot\cdot k+1} \models r_1$ implies that there exists a pair (p_i^1, s_i^1) in $\mathsf{LF}(r_1)$ such that $\sigma(0) \models p_i^1$ and $\sigma^{1\cdot\cdot k+1} \models s_i^1$. Since we know that $\sigma^{k\cdot\cdot} \models r_2$, it follows that $\sigma^{1\cdot\cdot} \models s_i^1 : r_2$. By definition $(p_i^1, s_i^1 : r_2)$ belongs to $\mathsf{LF}(r_1) : r_2$ which itself belongs to $\mathsf{LF}(r_1 : r_2)$. Therefore $\sigma \in \mathscr{L}(\mathsf{LF}(r_1 : r_2))$.

($\Longrightarrow$) Consider σ in $\mathscr{L}(\mathsf{LF}(r_1 : r_2))$. Looking at $\mathsf{LF}(r_1 : r_2)$ in Definition 8, either σ is matched by the left part of the union, or by the right part.

If it is matched by the left part, there exists a pair $(p_i^1, s_i^1 : r_2)$ such that $\sigma(0) \models p_i^1$ and $\sigma^{1\cdot\cdot} \models s_i^1 : r_2$. The latter implies that there exists $k \geq 1$ such that $\sigma^{1\cdot\cdot k+1} \models s_i^1$ and $\sigma^{k\cdot\cdot} \models r_2$. Therefore we have $\sigma^{0\cdot\cdot k+1} \models r_1$ and $\sigma^{k\cdot\cdot} \models r_2$, which implies $\sigma \in \mathscr{L}(r_1 : r_2)$.

If it is matched by the right part, there exists a pair $(p_i^1 \wedge p_j^2, s_j^2) \in \mathsf{LF}(r_1{:}r_2)$ such that $\sigma(0) \models p_i^1$, $\lambda(s_i^1) = \varepsilon$, $\sigma(0) \models p_j^2$, and $\sigma^{1\cdot\cdot} \models s_j^2$. The first two constraints imply that $\sigma^{0\cdot\cdot 1} \models r_1$, and the latter two that $\sigma^{1\cdot\cdot} \models r_2$. It follows that $\sigma \models r_1 : r_2$.

Other operators can be proven similarly. □

3.3. Automaton Construction

The traditional way to construct a finite automaton from such a linear form is to associate its states to regular expressions. For a state r, we interpret the pairs (p_i, s_i) in $\mathsf{LF}(r)$ as a transition $r \xrightarrow{p_i} s_i$. Algorithm 2 shows a

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

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

Algorithm 2: (baseline) Translation of a SERE ϕ to a NFA. Remember that rules (A)–(F) are always applied.

straightforward implementation of that construction. Final states are those that correspond to expressions that accept the empty word.

Let Terms(ϕ) designates the set of SEREs that appear in Q at the end of Algorithm 2. This set can be defined using the following recursive definition.

Definition 9 (Terms)**.** *For* $r \in \mathsf{SERE}$*, let* Terms(r) *denote the smallest subset of* SERE *such that* $r \in$ Terms(r) *and for each* $\phi \in$ Terms(r) *and each* $(p_i, s_i) \in \mathsf{LF}(\phi)$ *we have* $s_i \in$ Terms(r)*.*

Assuming that $|Q| = |\text{Terms}(\phi)|$ is finite, which we shall prove in the next section, the fact that $\mathscr{L}(\langle Q, \delta, \phi, F\rangle) = \mathscr{L}(\phi)$ follows from Theorem 1.

Although Terms(ϕ) is defined as the smallest subset of SERE recursively produced by LF, the resulting automaton is not necessarily minimal in terms of number of states. Since we only use syntactic equivalence, the construction can produce two states labeled by SEREs $r_1 \not\equiv r_2$ such that $\mathscr{L}(r_1) = \mathscr{L}(r_2)$.

This algorithm can be altered in several ways in attempt to simplify the resulting automaton. Section 4 discusses ways to *simplify* the linear forms $\boxed{\mathsf{LF}(f)}$ before they get used in Algorithm 2. Section 5 proposes larger modifications of Algorithm 2 meant to fuse states with identical linear forms.

3.4. Proof of Termination of Algorithm 2

To ensure termination of our translation algorithm we should prove that Terms(r) is finite: this is the upcoming Theorem 2. Our proof is inspired by a similar theorem by Antimirov [2, Theorem 3.4], however the results differ because of the new operators we support. Specifically, Antimirov did not support operators '$\wedge$', ':', and 'fm'.

Like Antimirov, we start by introducing a notion of *partial derivative* [2, Definition 2.8] which we adapt to our SEREs, where the alphabet is $\Sigma = 2^{AP}$.

Definition 10 (partial derivative)**.** *Given an expression $r \in \mathsf{SERE}$ and a valuation $x \in \Sigma = 2^{AP}$, we denote $\partial_x r$ the partial derivative of r with respect to x defined by:*

$$\partial_x r = \{s \mid (p, s) \in \mathsf{LF}(r),\, x \models p\}$$

We extend partial derivatives to the case $\partial_x R$ where $R \subseteq \mathsf{SERE}$ is a set of expression, in a natural way: $\partial_x R = \cup_{r \in R} \partial_x r$. We also extend the notation to support derivation by a nonempty word $w \in \Sigma^+$ with $\partial_w r = \partial_{w^{1..}} \partial_{w(0)} r$.

Furthermore, we write $\partial_{\Sigma^+} r = \bigcup_{w \in \Sigma^+} \partial_w r$ for the set of all partial derivatives one can obtain using nonempty words of any length.

Using the above notation, we have $\text{Terms}(r) = \partial_{\Sigma^+} r \cup \{r\}$.

Let us extend our Definition 4 of the constant term λ so it also works on a set $R \subseteq \mathsf{SERE}$ of expressions with $\lambda(R) = \bigvee_{r \in R} \lambda(r)$.

For $x \in \Sigma$, the following equalities follow from the Definitions 8 and 10:

$$\partial_x \bot = \varnothing \tag{1}$$

$$\partial_x \varepsilon = \varnothing \tag{2}$$

$$\partial_x b = \begin{cases} \{\varepsilon\} & \text{if } x \models b \\ \varnothing & \text{else} \end{cases} \tag{3}$$

$$\partial_x (r_1 \vee r_2) = \partial_x r_1 \cup \partial_x r_2 \tag{4}$$

$$\partial_x r^\star = (\partial_x r_1)\,;\, r^\star \tag{5}$$

$$\partial_x (r_1\,;\, r_2) = ((\partial_x r_1)\,;\, r_2) \cup (\lambda(r_1)\,;\, \partial_x r_2) \tag{6}$$

$$\partial_x (r_1 : r_2) = ((\partial_x r_1) : r_2) \cup (\lambda(\partial_x r_1)\,;\, \partial_x r_2) \tag{7}$$

$$\partial_x (r_1 \wedge r_2) = \{p_1 \wedge p_2 \mid p_1 \in \partial_x r_1,\, p_2 \in \partial_x r_2\} \tag{8}$$

$$\partial_x \mathsf{fm}(r) = \left\{ \bigvee_{s \in \partial_x r} s \right\} \tag{9}$$

Definition 11 (Suffix set)**.** *Given a nonempty word $w \in \Sigma^+$, let $Sfx(w)$ denote the set $\{w^{i..} \mid 0 \leq i < |w|\}$ of nonempty suffixes of w.*

Lemma 1. *For $w \in \Sigma^+$ the following (in)equalities follow from* (1)–(9)*:*

$$\partial_w(r_1 \vee r_2) = (\partial_w r_1) \cup (\partial_w r_2) \tag{10}$$

$$\partial_w r^\star \subseteq \bigcup_{v \in Sfx(w)} (\partial_v r) \,;\, r^\star \tag{11}$$

$$\partial_w(r_1 \,;\, r_2) \subseteq ((\partial_w r_1) \,;\, r_2) \cup \bigcup_{v \in Sfx(w)} \partial_v r_2 \tag{12}$$

$$\partial_w(r_1 : r_2) \subseteq ((\partial_w r_1) : r_2) \cup \bigcup_{v \in Sfx(w)} \partial_v r_2 \tag{13}$$

$$\partial_w(r_1 \wedge r_2) = \{p_1 \wedge p_2 \mid p_1 \in \partial_w r_1,\, p_2 \in \partial_w r_2\} \tag{14}$$

$$\partial_w \mathsf{fm}(r_1) = \left\{ \bigvee_{s \in \partial_w r} s \right\} \tag{15}$$

Proof. Equations (10)–(12) are already known results [2, Lemma 3.3].

Let us prove (12) again, using our notations. Deriving $r_1 \,;\, r_2$ using (6) and words of increasing lengths, we have the following:

$$\begin{aligned}
\partial_a(r_1 \,;\, r_2) &= ((\partial_a r_1) \,;\, r_2) \cup (\lambda(r_1) \,;\, (\partial_a r_2)) && \text{for } a \in \Sigma \\
\partial_{ab}(r_1 \,;\, r_2) &= ((\partial_{ab} r_1) \,;\, r_2) \cup (\lambda(\partial_a r_1) \,;\, (\partial_b r_2)) \cup (\lambda(r_1) \,;\, (\partial_{ab} r_2)) && \text{for } ab \in \Sigma^2 \\
\partial_{abc}(r_1 \,;\, r_2) &= ((\partial_{abc} r_1) \,;\, r_2) \cup (\lambda(\partial_{ab} r_1) \,;\, (\partial_c r_2)) && \text{for } abc \in \Sigma^3 \\
&\quad \cup (\lambda(\partial_a r_1) \,;\, (\partial_{bc} r_2)) \\
&\quad \cup (\lambda(r_1) \,;\, (\partial_{abc} r_2))
\end{aligned}$$

Since those $\lambda(r_1)$, $\lambda(\partial_a r_1)$, $\lambda(\partial_{ab} r_1)$... are used to conditionally enable the subsequent terms, it should be clear that for any word $w \in \Sigma^+$, the set $\partial_w(r_1 \,;\, r_2)$ contains $((\partial_w r_1) \,;\, r_2)$ and a subset of $\bigcup_{v \in Sfx(w)} \partial_v r_2$, justifying equation (12).

Equation (13) is proven similarly:

$$\begin{aligned}
\partial_a(r_1 : r_2) &= ((\partial_a r_1) : r_2) \cup (\lambda(\partial_a r_1)\,;(\partial_a r_2)) && \text{for } a \in \Sigma\\
\partial_{ab}(r_1 : r_2) &= ((\partial_{ab} r_1) : r_2) \cup (\lambda(\partial_{ab} r_1)\,;(\partial_b r_2)) && \text{for } ab \in \Sigma^2\\
&\quad \cup (\lambda(\partial_a r_1)\,;(\partial_{ab} r_2))\\
\partial_{abc}(r_1 : r_2) &= ((\partial_{abc} r_1) : r_2) \cup (\lambda(\partial_{abc} r_1)\,;(\partial_c r_2)) && \text{for } abc \in \Sigma^3\\
&\quad \cup (\lambda(\partial_{ab} r_1)\,;(\partial_{bc} r_2))\\
&\quad \cup (\lambda(\partial_a r_1)\,;(\partial_{abc} r_2))
\end{aligned}$$

Finally, equations (14)–(15) follow immediately from (8)–(9). □

Lemma 2. *For two SEREs r_1 and r_2 we have:*

$$\begin{aligned}
|\partial_{\Sigma^+}(r_1 \vee r_2)| &\leq |\partial_{\Sigma^+} r_1| + |\partial_{\Sigma^+} r_2|\\
|\partial_{\Sigma^+} r_1^\star| &\leq |\partial_{\Sigma^+} r_1|\\
|\partial_{\Sigma^+}(r_1\,;r_2)| &\leq |\partial_{\Sigma^+} r_1| + |\partial_{\Sigma^+} r_2|\\
|\partial_{\Sigma^+}(r_1 : r_2)| &\leq |\partial_{\Sigma^+} r_1| + |\partial_{\Sigma^+} r_2|\\
|\partial_{\Sigma^+}(r_1 \wedge r_2)| &\leq |\partial_{\Sigma^+} r_1| \times |\partial_{\Sigma^+} r_2|\\
|\partial_{\Sigma^+}\mathsf{fm}(r_1)| &\leq 2^{|\partial_{\Sigma^+} r_1|}
\end{aligned}$$

Proof. These inequalities are consequences of equations (10)–(15) and Definition 10.

$$\begin{aligned}
|\partial_{\Sigma^+}(r_1 \vee r_2)| &= \left|\bigcup_{w\in\Sigma^+} \partial_w(r_1 \vee r_2)\right| && \text{by Def. 10}\\
&= \left|\bigcup_{w\in\Sigma^+} (\partial_w r_1) \cup (\partial_w r_2)\right| && \text{by (10)}\\
&= |(\partial_{\Sigma^+} r_1) \cup (\partial_{\Sigma^+} r_2)| && \text{by Def. 10}\\
&\leq |\partial_{\Sigma^+} r_1| + |\partial_{\Sigma^+} r_2|\\
|\partial_{\Sigma^+} r_1^\star| &= \left|\bigcup_{w\in\Sigma^+} \partial_w r_1^\star\right| \leq \left|\bigcup_{w\in\Sigma^+}\bigcup_{v\in Sfx(w)} (\partial_v r_1)\,;r_1^\star\right| && \text{by Def. 10 and (11)}\\
&\leq \left|\bigcup_{w\in\Sigma^+} (\partial_w r_1)\,;r_1^\star\right| \overset{\text{Def. 10}}{=} |(\partial_{\Sigma^+} r_1)\,;r_1^\star| = |\partial_{\Sigma^+} r_1|
\end{aligned}$$

$$
\begin{aligned}
|\partial_{\Sigma^+}(r_1\,;r_2)| &= \left|\bigcup_{w\in\Sigma^+}\partial_w(r_1\,;r_2)\right| && \text{by Def. 10}\\
&\le \left|\bigcup_{w\in\Sigma^+}\left(((\partial_w r_1)\,;r_2)\cup\bigcup_{v\in Sfx(w)}\partial_v r_2\right)\right| && \text{by (12)}\\
&= \left|\left(\bigcup_{w\in\Sigma^+}(\partial_w r_1)\,;r_2\right)\cup\bigcup_{w\in\Sigma^+}\partial_w r_2\right|\\
&= |((\partial_{\Sigma^+} r_1)\,;r_2)\cup(\partial_{\Sigma^+} r_2)| && \text{by Def. 10}\\
&\le |\partial_{\Sigma^+} r_1| + |\partial_{\Sigma^+} r_2|
\end{aligned}
$$

The proof that $|\partial_{\Sigma^+}(r_1 : r_2)| \le |\partial_{\Sigma^+} r_1| + |\partial_{\Sigma^+} r_2|$ is exactly the same as above, using (13) instead of (12).

$$
\begin{aligned}
|\partial_{\Sigma^+}(r_1\wedge r_2)| &= \left|\bigcup_{w\in\Sigma^+}\partial_w(r_1\wedge r_2)\right| && \text{by Def. 10}\\
&= \left|\bigcup_{w\in\Sigma^+}\{p_1\wedge p_2 \mid p_1\in\partial_w r_1,\, p_2\in\partial_w r_2\}\right| && \text{by (14)}\\
&\le |\{p_1\wedge p_2 \mid p_1\in\partial_{\Sigma^+} r_1,\, p_2\in\partial_{\Sigma^+} r_2\}|\\
&\le |\partial_{\Sigma^+} r_1|\times|\partial_{\Sigma^+} r_2|
\end{aligned}
$$

$$
\begin{aligned}
|\partial_{\Sigma^+}\mathsf{fm}(r_1)| &= |\bigcup_{w\in\Sigma^+}\partial_w\mathsf{fm}(r_1)| && \text{by Def. 10}\\
&= \left|\left\{\mathsf{fm}\Big(\bigvee_{p\in\partial_w r_1} p\Big)\;\middle|\; w\in\Sigma^+\right\}\right| && \text{by (15)}\\
&\le \left|\left\{\mathsf{fm}\Big(\bigvee_{p\in P} p\Big)\;\middle|\; P\in\partial_{\Sigma^+}r_1\right\}\right| = 2^{|\partial_{\Sigma^+}r_1|}
\end{aligned}
$$

□

Corollary 1. *For any expression r, the set $\partial_{\Sigma^+}r$ is finite.*

Proof. Straightforward induction on the grammar of r, using Lemma 2. □

Theorem 2. *For any $r \in \mathsf{SERE}$,* $\mathrm{Terms}(r)$ *is finite.*

Proof. Since $\mathrm{Terms}(r) = \partial_{\Sigma^+}r \cup \{r\}$, this follows from corollary 1. □

From the inequalities in Lemma 2, one can observe that the added operators do not have the same cost. In particular $\wedge$ incurs a quadratic cost, while fm leads to an exponential blow up.

Corollary 2. *For a SERE r, let $\|r\|$ denote the number of occurrences of maximal Boolean subformulas in r (i.e., the number of times rule "b" was used to produce r with the grammar given in Definition 2).*

1. *If r does not use operators $\wedge$ and fm, we have $|\partial_{\Sigma^+}r| \le \|r\|$.*
   (In other words, adding the ":" operator to the set of classical regular operators preserves the bound established by Antimirov [2, Theorem 3.4].)
2. *If r does not use operator fm, we have $|\partial_{\Sigma^+}r| \le 2^{\|r\|}$.*

Proof. Straightforward induction on the grammar of r, using Lemma 2. □

4. Linear Form Simplifications

When constructing an automaton from a linear form, it is possible to alter the shape of the automaton constructed by transforming the linear forms it uses into other, equivalent linear forms. In this section we present a few transformations that aim at simplifying linear forms. By "*simplifying*" we

mean to reduce the number of pairs in the hope that this results in a smaller automaton. Simplifying a finite automaton can of course be done after its construction using more traditional algorithms like bisimulation-based reductions [23], however it is always good to look for cheap opportunities to keep the intermediate automaton small.

Definition 12 (Unique Suffix and Unique Prefix simplifications)**.** *Let L be a linear form, let* $\mathsf{MergePre}(L, s) = \bigvee_{(p,s)\in L} p$ *be the (Boolean) disjunction of all prefixes sharing a given suffix s, and let* $\mathsf{MergeSuf}(L, p) = \bigvee_{(p,s)\in L} s$ *be the (rational) disjunction of all suffixes sharing a given prefix p.*

We define US *(unique suffixes) and* UP *(unique prefixes) as follows:*

$$\mathsf{US}(L) = \{(\mathsf{MergePre}(L, s), s) \mid (p, s) \in L\}$$
$$\mathsf{UP}(L) = \{(p, \mathsf{MergeSuf}(L, p)) \mid (p, s) \in L\}$$

Replacing $\mathsf{LF}(f)$ by $\mathsf{US}(\mathsf{LF}(f))$ in Algorithm 2 is equivalent to merging the edges of the automaton that have the same source and same destination. For instance $\mathsf{US}(\{(a, r), (b, r)\}) = \{(a \vee b, r)\}$.

Replacing $\mathsf{LF}(f)$ by $\mathsf{UP}(\mathsf{LF}(f))$ in Algorithm 2 is merging outgoing edges that share the same label. In Antimirov's setup [2], where prefixes of linear forms are letters, using UP would create a deterministic automaton. However, because in our setup prefixes are Boolean formulas, this is not the case: UP can remove *some* non-determinism, but the result will not necessarily be deterministic. For instance the non-deterministic linear form $\{(a, q_1), (a\wedge b, q_2)\}$ is unchanged by UP. If we wish to construct a deterministic automaton, we can use $\mathsf{det}(\mathsf{LF}(f))$ (see Proposition 1). Our intent with UP is therefore not to produce a deterministic automaton, but to help reduce the size of a non-deterministic result.

We should point out that the equivalent of Theorem 9 still holds when $\mathsf{UP}(\mathsf{LF}(f))$ is used because the terms created by this new variant are disjunctions of terms created by the original construction.

Unfortunately, it is also possible that using UP will introduce new additional states in the automaton. For instance $\mathsf{UP}(\{(a, q_1), (a, q_2), (b, q_1), (\neg b, q_2)\}) = \{(a, q_1 \vee q_2), (b, q_1), (\neg b, q_2)\}$ would be introducing the state $q_1 \vee q_2$ that was not present initially.

Because US only merges edges, it sounds natural to use it together with UP. However, while it is possible to find cases where replacing $\mathsf{LF}(f)$ by $\mathsf{US}(\mathsf{UP}(\mathsf{LF}(f)))$ is better than replacing $\mathsf{LF}(f)$ by $\mathsf{UP}(\mathsf{US}(\mathsf{LF}(f)))$, the opposite also exists. For instance, figure 2 shows one case where replacing

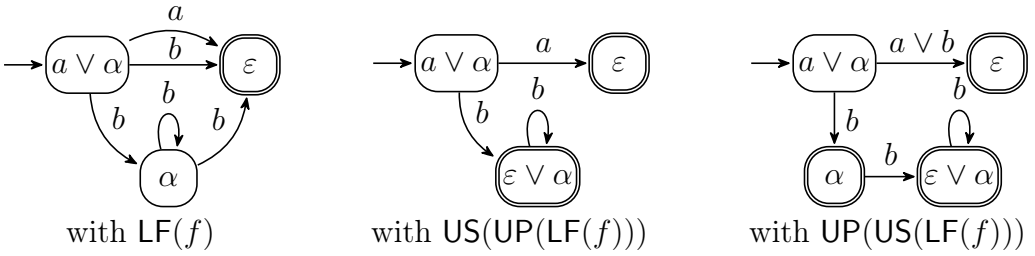

Figure 2: Three equivalent automata built with variants of Algorithm 2 for the formula $a \vee \alpha$ where α abbreviates $b^\star : b$. (Note that α is equivalent to b^+ but the algorithm does not know that.) We have $\mathsf{LF}(\alpha) = \{(b, \varepsilon), (b, \alpha)\}$ so it follows that $\mathsf{LF}(a \vee \alpha) = \{(a, \varepsilon), (b, \varepsilon), (b, \alpha)\}$. Applying UP first on the initial formula does not leave anything for US to simplify: $\mathsf{US}(\mathsf{UP}(\mathsf{LF}(a \vee \alpha))) = \mathsf{UP}(\mathsf{LF}(a \vee \alpha)) = \{(a, \varepsilon), (b, \varepsilon \vee \alpha)\}$. Conversely, applying US first makes UP useless in this case: $\mathsf{UP}(\mathsf{US}(\mathsf{LF}(a \vee \alpha))) = \mathsf{US}(\mathsf{LF}(a \vee \alpha)) = \{(a \vee b, \varepsilon), (b, \alpha)\}$.

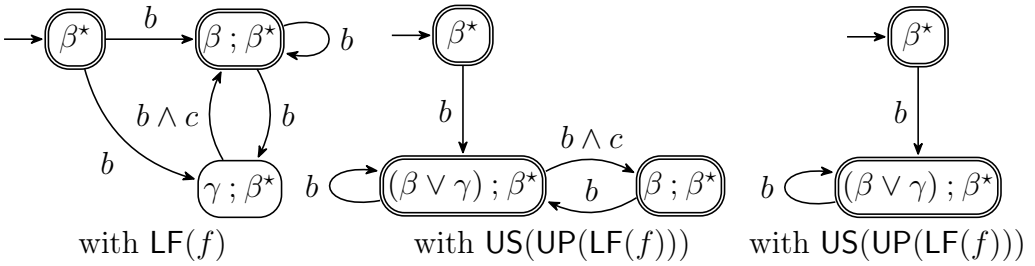

Figure 3: Three equivalent automata built with variants of Algorithm 2 for the formula $\beta^\star$ where $\beta = (b^\star \wedge (1 \vee (1 ; c))^\star)$. We also use $\gamma = (b^\star \wedge (c ; ((1 \vee (1 ; c))^\star)))$. We have $\mathsf{LF}(\beta^\star) = \{(b, \beta ; \beta^\star), (b, \gamma ; \beta^\star)\}$, $\mathsf{LF}(\beta ; \beta^\star) = \mathsf{LF}(\beta^\star)$, and $\mathsf{LF}(\gamma ; \beta^\star) = \{(b \wedge c, \beta ; \beta^\star)\}$. The order of US and UP does not change anything when processing the initial state; we have $\mathsf{US}(\mathsf{UP}(\mathsf{LF}(\beta^\star))) = \mathsf{UP}(\mathsf{US}(\mathsf{LF}(\beta^\star))) = \{(b, (\beta \vee \gamma) ; \beta^\star)\}$. However it does have an influence when processing $(\beta \vee \gamma) ; \beta^\star$. Let us call $L = \mathsf{LF}((\beta \vee \gamma) ; \beta^\star) = \{(b, \beta ; \beta^\star), (b, \gamma ; \beta^\star), (b \wedge c, \beta ; \beta^\star)\}$. Then $\mathsf{US}(\mathsf{UP}(L)) = \mathsf{UP}(L) = \{(b, (\beta \vee \gamma) ; \beta^\star), (b \wedge c, \beta ; \beta^\star)\}$ but $\mathsf{US}(L) = \{(b, \beta ; \beta^\star), (b, \gamma ; \beta^\star)\}$ and therefore $\mathsf{US}(L) = \{(b, (\beta \vee \gamma) ; \beta^\star)\}$.

$\mathsf{LF}(f)$ by $\mathsf{US}(\mathsf{UP}(\mathsf{LF}(f)))$ in Algorithm 2 produces a smaller automaton than when using $\mathsf{UP}(\mathsf{US}(\mathsf{LF}(f)))$. Figure 3 shows an opposite case, where using $\mathsf{US}(\mathsf{UP}(\mathsf{LF}(f)))$ results in a larger automaton than with $\mathsf{UP}(\mathsf{US}(\mathsf{LF}(f)))$.

5. Signature and Transition-Based Variants

We now discuss variants of Algorithm 2. Those variants can also benefit from the previous simplifications.

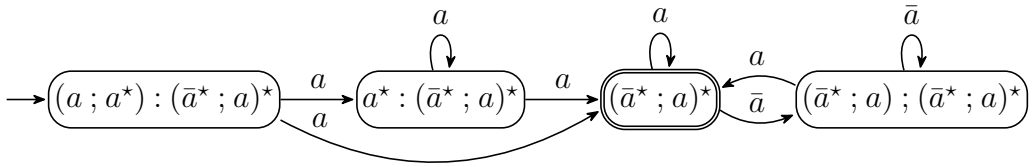

Figure 4: Automaton for $\varphi = (a\,;a^\star):(\bar{a}^\star\,;a)^\star$. We have $\mathsf{LF}(\varphi) = \mathsf{LF}(a^\star:(\bar{a}^\star\,;a)^\star) = \{(a, a^\star:(\bar{a}^\star\,;a)^\star), (a,(\bar{a}^\star\,;a)^\star)\}$, and $\mathsf{LF}((\bar{a}^\star\,;a)^\star) = \mathsf{LF}((\bar{a}^\star\,;a)\,;(\bar{a}^\star\,;a)^\star) = \{(a,(\bar{a}^\star\,;a)^\star), (\bar{a},(\bar{a}^\star\,;a)\,;(\bar{a}^\star\,;a)^\star)\}$.

Consider the automaton of Figure 4, where the first two states are labeled by formulas that have the same linear form, and so are the last two states. If two expressions r_1 and r_2 have the same linear form $\mathsf{LF}(r_1) = \mathsf{LF}(r_2)$, it implies that $\mathscr{L}(r_1) \setminus \varepsilon = \mathscr{L}(r_2) \setminus \varepsilon$. Therefore, states that correspond to formulas with the same linear form (i.e., states that have identical sets of outgoing transitions) can be merged if they are both accepting, or both rejecting. Thus, the first two states of Figure 4 could be merged.

We can obtain such a merge automatically if we modify our translation as in Algorithm 3 to label each state by a pair $(\mathsf{LF}(\varphi), \lambda(\varphi))$ that we call the *signature* of φ. This gives the automaton of Figure 5.

Currently, the last two states of Figure 5 may not be merged because one is accepting while the other is not. We could however merge them by changing our automaton formalism such that the notion of acceptance is carried by the transitions instead of the states. Although finite automata are seldom used with transition-based acceptance [27], ω-automata (i.e., automata over infinite words) with transition-based acceptance have been used for a long time as they often lead to simpler algorithms [25, 21, 22, 20, 26, to cite a few]. Let us define a transition-based finite automaton:

Definition 13 (TFA)**.** *A* transition-based finite automaton *is a tuple* $\mathcal{A} = \langle Q, \delta, \iota, \beta\rangle$ *where* Q *is a finite set of states,* $\delta \subseteq Q \times \mathbb{B}(AP) \times \mathbb{B} \times Q$ *is the transition relation,* $\iota \in Q$ *is the initial state, and* $\beta \in \mathbb{B}$ *is a Boolean indicating whether* ε *should be accepted.*

We write $s \xrightarrow{f,b} d$ *when* $(s, f, b, d) \in \delta$.

A sequence of valuations $\sigma \in \Sigma^n$ *of size* n *is* accepted *by* $\mathcal{A}$ *if either* $n = 0$ *and* $\beta = \top$*, or* $n > 0$ *and there exists a sequence of transitions* $\rho = s_0 \xrightarrow{f_0,b_0} s_1 \xrightarrow{f_1,b_1} \cdots \xrightarrow{f_{n-1},b_{n-1}} s_n$ *such that* $s_0 = \iota$*,* $b_{n-1} = \top$*, and for all* i*,* $\sigma(i) \models f_i$.

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

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

Algorithm 3: (sig.-based) Translation that identifies states with identical linear form and identical ε acceptance.

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

$L_\iota, b_\iota \leftarrow \mathsf{LF}(\phi), [\lambda(\phi) = \varepsilon]$
$Q, \delta \leftarrow \{L_\iota\}, \varnothing$
todo.push(L_ι)
while todo $\neq \varnothing$ **do**
  $L \leftarrow$ todo.pop()
  foreach $(p, s) \in L$ **do**
    $L', b' \leftarrow \mathsf{LF}(s), [\lambda(s) = \varepsilon]$
    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 4: (trans.-based) Translation to transition-based automata, identifying states with identical linear form regardless of ε acceptance.

The language *of $\mathcal{A}$, denoted $\mathscr{L}(\mathcal{A})$ is the set of words accepted by $\mathcal{A}$.*

In other words, transitions of a TFA carry an extra Boolean that is used to mark the transition as accepting, and a word is accepted if it is recognized by a run whose last transition is accepting. Graphically, we represent accepting transitions using arrows with double lines. The acceptance of the empty word is indicated by a special Boolean β in the definition, and can be represented graphically by using double lines on the arrow indicating the initial state.

TFAs enjoy properties similar to traditional finite automata: they are as expressive as regular expressions, and are closed under Boolean operations [27]. However, they can be slightly smaller, as we shall see in our evaluation. For instance, Algorithm 4 generates the TFA of Figure 6, which is equivalent to the NFA of Figure 5.

We have additional motivation for using TFAs. The reason we are working on translating SERE to automata is that SEREs are part of the PSL

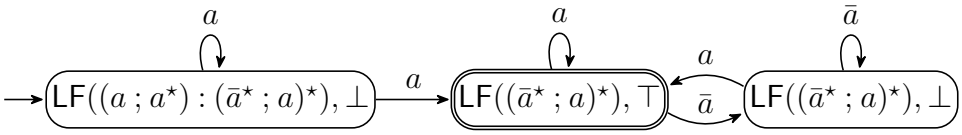

Figure 5: Automaton obtained by merging states labeled by formulas that have the same linear form and the same acceptance of ε.

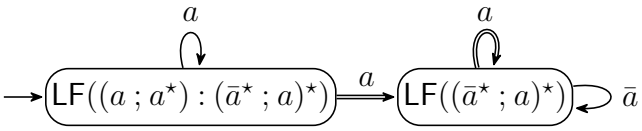

Figure 6: Transition-based automaton obtained by merging states labeled by formulas that have the same linear form, regardless of the acceptance of ε, since the latter is decided on transitions.

and SVA standards. However, the PSL/SVA standards assume a SERE will always match a non-empty word. Therefore, the Boolean β that we added to the definition of a TFA to allow it to recognize ε can simply be ignored. Furthermore, as we translate a PSL formula into an ω-automaton, we build upon translation algorithms that naturally produce transition-based ω-automata [13]. In this context, it seems more natural to have SERE converted into TFA. The fact that TFA are more succinct comes as a bonus.

6. Experimental Evaluation

A reproducibility package, archived at https://doi.org/10.5281/zenodo.15069061, contains our implementation, a Jupyter notebook to use it interactively, and scripts to reproduce our experiments.

6.1. Implementation Considerations

Our algorithms have been implemented in a development version of Spot [14].

Since all SEREs handled during translation are combinations of subformulas of the original SERE, we represent all SEREs as a direct acyclic graph in which each sub-formula has a unique representation. A node in this DAG stores an operator and a list of pointers to the nodes of its operands. A global unicity table allows to map a pair (operator, operands) to the unique node that represents it if it exists in the graph. Additionally, we apply the rules (A)–(F) (from page 4) during the construction of the node. In particular, for rule (A), we handle associative operators as n-ary operators, and we

inline their operands when they have the same top-level operator. Rule (C) is achieved by sorting all operands (for instance in their node creation order), and this then makes it easy to locate duplicate elements for rule (I1).

Linear forms, as introduced in Definition 6 are sets of pairs (p_i, s_i) where p_i is a Boolean formula which we represent as a BDD, and s_i is a SERE. Several data structures could be used to represent a set of such pairs, and the choice of the data structure could also depend on whether we later plan to use UP or US. For instance if one plans to use UP, it is tempting to represent linear forms as hash maps that map each p_i to its s_i, so that the latter can be updated whenever a new pair (p_i, s'_i) is introduced into the linear form. However, during development, we noticed that linear forms were usually very small (average 3.12, median 1), making the construction of such mappings more expensive than simpler data structures. As a consequence, we simply represent them as arrays of pairs. If UP or US needs to be applied, we first sort those pairs by prefixes or suffixes, and then merge the relevant pairs.

The results presented hereafter differ from those presented at CIAA'24 [24] because we uncovered a flaw in the implementation while preparing this version. In our previous implementation, the array of pairs representing the linear form could have duplicate pairs and were not sorted: these duplicate pairs appeared as extra edges, prevented some signature matches, and made the effect of UP and US much stronger than it really is. In the current implementation, the vector is also sorted when UP or US is not used in order to remove duplicate pairs.

6.2. SERE Benchmark Random Generation

We are not aware of any existing benchmark of industrial-grade SEREs or PSL formulas. The PROSYD project built a handbook of many PSL examples formulas [5], unfortunately most of their SEREs are very small. To test their PSL translation, Cimatti et al. [11] reused the above examples, but replacing all SEREs by random ones to make the formulas more difficult.

In the absence of any better source for SEREs, we resort to evaluating our work on randomly generated inputs; this is probably not representative of realistic use-cases of SEREs. We generated random SERE usings Spot's `randltl` tool, with equal probability of occurrence for all SERE operators. To ensure a diversity of SERE length and number of atomic propositions, we grouped the random SEREs into bins of expressions with equal size (number of nodes in their syntax tree) and equal number of unique atomic propositions, capping each bin to 50 expressions. The resulting set has 12500

unique SEREs with sizes ranging from 1 to 35, and between 1 and 15 atomic propositions.

6.3. Results

Benchmarks were run on an AMD Ryzen 5 3600 CPU, with 16GB of RAM, and with core frequency capped to 2.2GHz to minimize the impact of throttling on timing measurements. For each SERE, we evaluated variants of the translation by measuring the number of states and edges of the produced automata, as well as the time needed to produce them.

All scales are logarithmic. Scatter plots that show number of states use a jitter of ± 0.4 over their position to distinguish points. The numbers in the top left and bottom right corners of the plots specify how many points are strictly above or below the diagonal.

Plots refer to Algorithm 2, 3 and 4 as Baseline, Sig and Trans respectively.

Figure 7 compares these three algorithms in terms of number of states, number of edges, and run time. We can observe that, with respect to the number of states and edges, Sig improves upon Baseline, and Trans improves upon Sig. The 620 cases where Trans improves upon Baseline represent 5% of the set of 12500 formulas tested. Time-wise, the three approaches seem hard to compare because of the large amount of very quick cases, where time measurement is inherently noisy. Looking at the cases with a larger runtime, it seems that using signatures causes a slowdown over the baseline, but the transition-based version compensates this slowdown and has a runtime comparable to the baseline algorithm.

We checked the determinism of the automata produced by the three translations. Out of the 12500 automata we produced, only 2448 (19.6%) are non-deterministic. This number is the same for all three algorithms: the benchmark does not contain any case where merging states that have equivalent signatures turns a non-deterministic automaton into a deterministic one.

The effect of UP on the three algorithms is shown by Figure 8. Bear in mind that UP may only have an effect on non-deterministic automata. However, when translating a SERE we do not know upfront if the result will be deterministic or not, so we present results on the whole dataset again. UP has a very mitigated results on Baseline: it improves the number of states of the automaton almost as often as it worsens it. The effect on slightly better on Sig and Trans. The number of edges follow a similar trend.

If we compare the 122 cases of the middle-right plot, where UP reduces

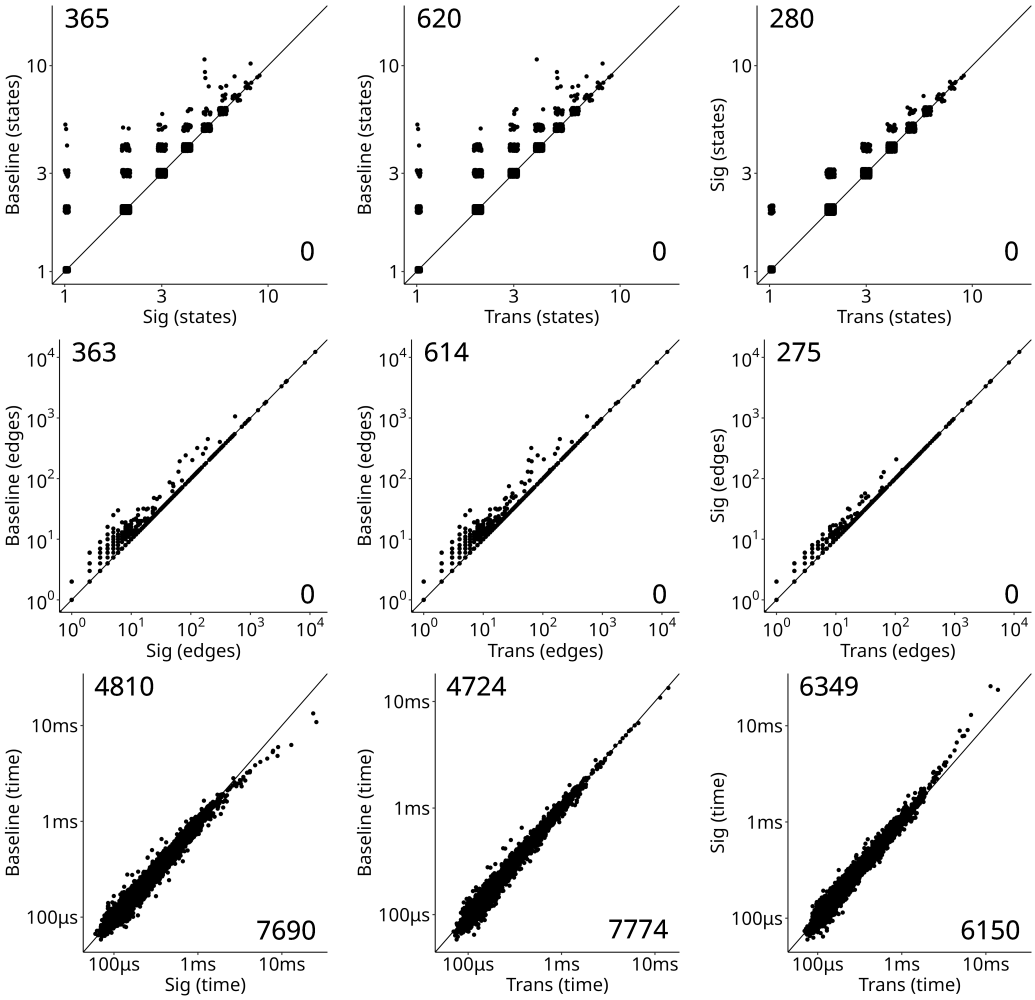

Figure 7: Comparisons of Algorithm 2 (Baseline), Algorithm 3 (Sig), and Algorithm 4 (Trans).

the number of edges for Algorithm 4, this represents 4.9% of the 2448 non-deterministic automata produced without UP.

In addition to the small reduction in states and edges, using UP reduces the number of non-deterministic automata produced by the three algorithms to 2409. This is a very small improvement, but the effect of using UP on run time is completely negligible: if UP is used, the implementation has to sort each linear form according to the prefix of its pairs, and then merge pairs with identical prefixes; if UP is not used, the implementation has to sort each

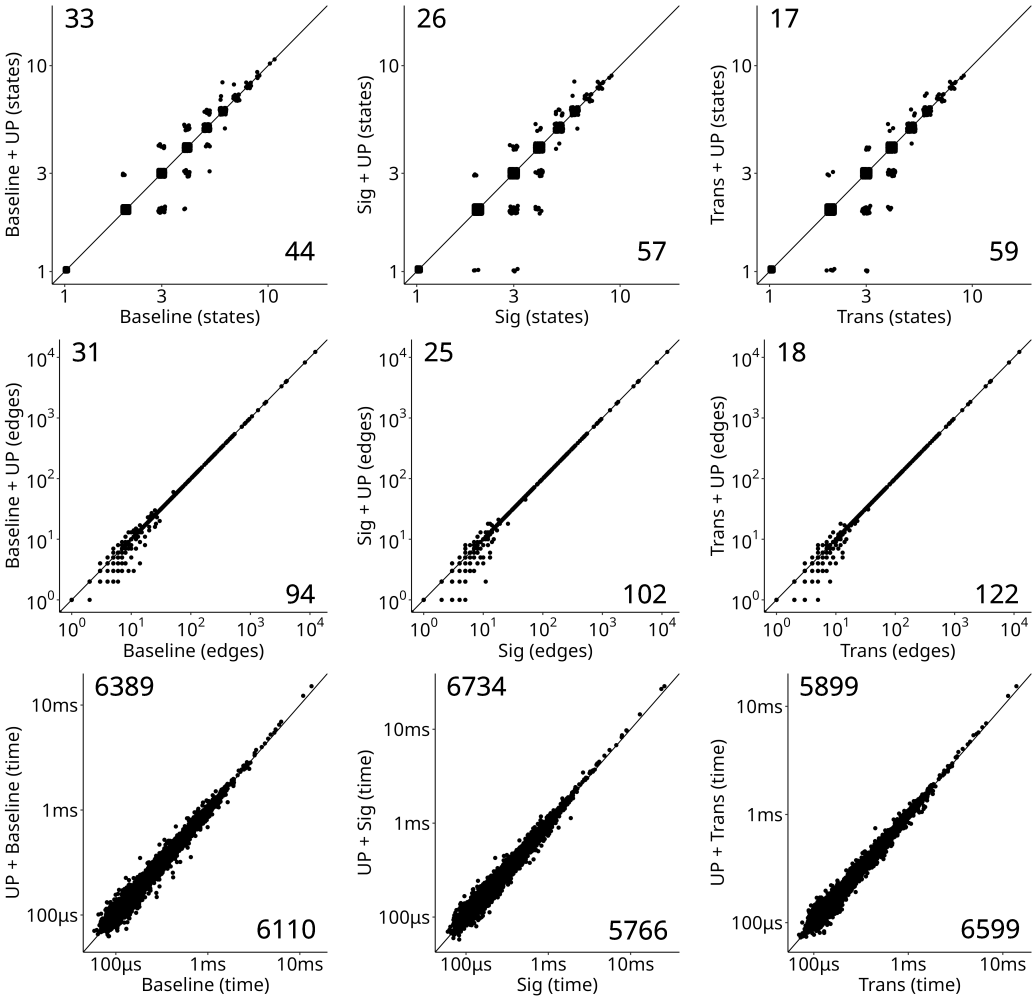

Figure 8: Effect of UP on the three algorithms.

linear form to detect duplicate pairs and remove them.

Figure 9 shows the effect of US. As expected, we see a large reduction of edges, regardless of the algorithm used. The runtime with or without US is almost unchanged, so this reduction in edges does not come at a significant computational cost.

Figure 10 shows that in practice, the impact of the order of application between UP and US discussed in Section 4 is rather limited, producing automata with a different number of states in only 20 cases out of our 12500 formulas. Since the difference is in favor of US $\circ$ UP, we show the effect

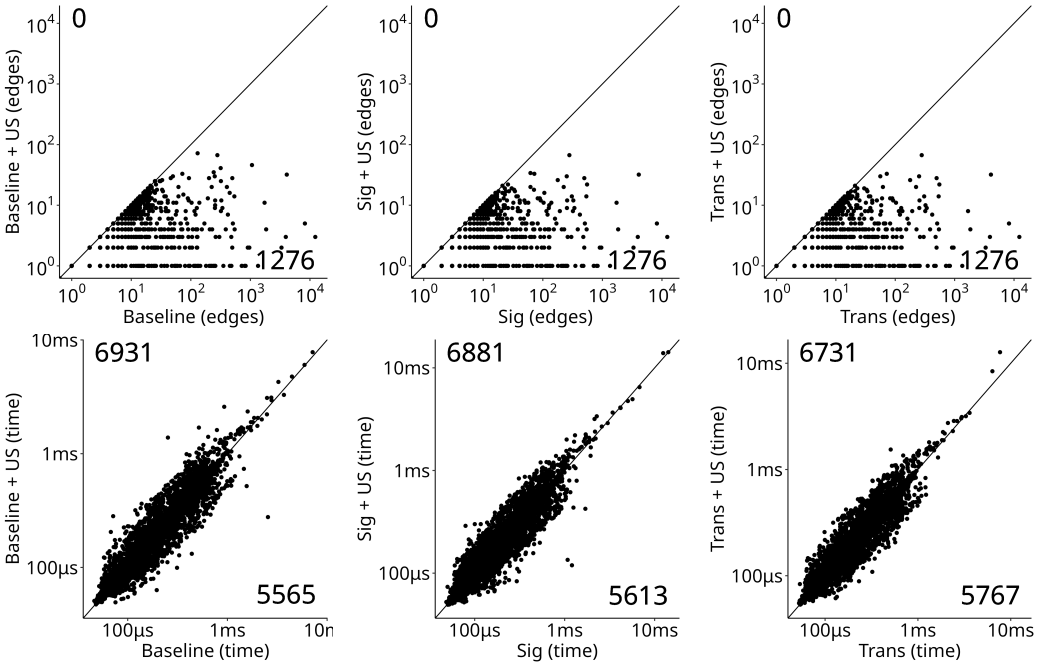

Figure 9: Effect of US on the three algorithms (states are not shown, because they are unaffected).

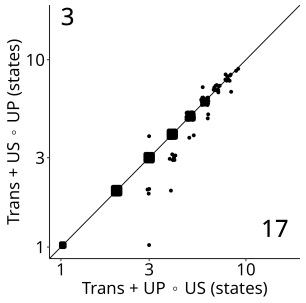

Figure 10: Comparison of UP ∘ US versus US ∘ UP

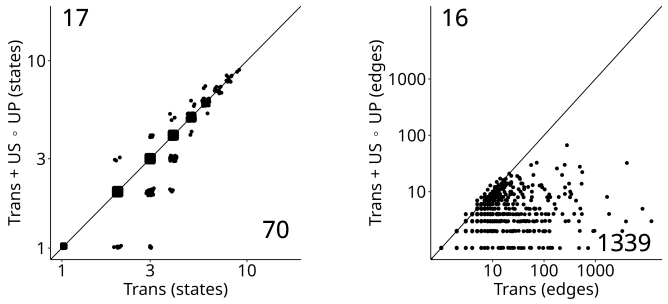

Figure 11: Impact of US ∘ UP on Algorithm 4

of that combination of Algorithm 4 in Figure 11: the number of states is marginally reduced, and the number of edges is very largely reduced. The latter reduction is expected, as US can only merge edges.

7. Conclusion

We adapted Antimirov's non-deterministic automata construction based on linear forms, to the semi-extended regular expressions used by the PSL and SVA standards.

To evaluate the translation, we constructed a SERE benchmark dataset, which we hope can be reused in future work to compare different SERE, PSL or SVA translators.

We also introduced alternative translation algorithms that use the linear form to simplify the automaton during its construction, or that build a transition-based automaton. While the new constructions can only construct smaller automata, and have a either negligible or no runtime overhead, the best translation (the transition-based one) improves only the size of 5% of the automata of our entire benchmark.

As these SERE are defined on alphabet of the form 2^{AP}, we introduced some rewritings (UP, US) of these linear forms. While it was clear that US would reduce the number of edges by merging them, we focused our evaluation on the effect of UP. The impact of UP turned out to be very limited with marginal improvement for no runtime costs. (These results differ from our previous work at CIAA'24 [24] because that previous implementation had an implementation flaw its baseline.)

Our conclusion is that using transition-based automata, labeling them with linear forms, and simplifying those linear forms with UP and then US are cheap and effective ways of keeping the output small. A compact output matters in applications where the automaton is constructed on-the-fly or only partially, and therefore cannot benefit from subsequent simplifications. Satisfiability, which cannot be decided syntactically because of the intersection operator, is one such problem.

In future work, we aim to integrate this SERE translation into a larger PSL translation procedure that produce Büchi automata with transition-based acceptance. This was our main motivation for building a transition-based SERE translation. We believe it would also make sense to reimplement linear forms using multi-terminal binary decision diagrams, as done by Duret-Lutz et al. [15] in the context of LTL_f synthesis.

References

1. *1800-2017 - IEEE Standard for SystemVerilog–Unified Hardware Design, Specification, and Verification Language*. IEEE, Feb. 2018. doi: 10.1109/IEEESTD.

2018.8299595.

2. V. Antimirov. Partial derivatives of regular expressions and finite automaton constructions. *Theoretical Computer Science*, 155(2):291–319, Mar. 1996. doi: 10.1016/0304-3975(95)00182-4.

3. C. Baier and J.-P. Katoen. *Principles of Model Checking*. MIT Press, 2008.

4. E. Bartocci, Y. Falcone, A. Francalanza, and G. Reger. *Introduction to Runtime Verification*, pages 1–33. Springer International Publishing, Cham, 2018. doi: 10.1007/978-3-319-75632-5_1.

5. S. Ben-David and A. Orni. "property-by-example" guide: a handbook of PSL examples. Technical Report 1.1/3, PROSYD, 2005. URL https://web.archive.org/web/20070207074155/http://www.prosyd.org/twiki/pub/Public/DeliverablePageWP1/prosyd1.1_3.pdf.

6. S. Broda, A. Machiavelo, N. Moreira, and R. Reis. On the equivalence of automata for KAT-expressions. In A. Beckmann, E. Csuhaj-Varjú, and K. Meer, editors, *Language, Life, Limits — Proceedings of the 10th Conference on Computability in Europe (CiE'14)*, pages 73–83, Cham, 2014. Springer International Publishing. ISBN 978-3-319-08019-2. doi: 10.1007/978-3-319-08019-2_8.

7. S. Broda, S. Cavadas, M. Ferreira, and N. Moreira. Deciding synchronous kleene algebra with derivatives. In F. Drewes, editor, *Proceedings of the 20th International Conference on Implementation and Application of Automata (CIAA'15)*, pages 49–62, Cham, 2015. Springer International Publishing. doi: 10.1007/978-3-319-22360-5_5.

8. R. E. Bryant. Symbolic boolean manipulation with ordered binary-decision diagrams. *ACM Computing Surveys*, 24(3):293–318, Sept. 1992.

9. J. A. Brzozowski. Derivatives of regular expressions. *Journal of the ACM*, 11 (4):481–494, Oct. 1964. doi: 10.1145/321239.321249.

10. P. Caron, J.-M. Champarnaud, and L. Mignot. Partial derivatives of an extended regular expression. In *Proceedings of the 5th International Conference on Language and Automata Theory and Applications (LATA'11)*, pages 179–191. Springer Berlin Heidelberg, 2011. doi: 10.1007/978-3-642-21254-3_13.

11. A. Cimatti, M. Roveri, and S. Tonetta. Syntactic optimizations for psl verification. In O. Grumberg and M. Huth, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 505–518, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg. doi: 10.1007/978-3-540-71209-1_39.

12. L. D'Antoni and M. Veanes. Minimization of symbolic automata. In S. Jagannathan and P. Sewell, editors, *Proceedings of the 41st Annual Symposium on Principles of Programming Languages (POPL'14)*, pages 541–554. ACM, Jan. 2014. doi: 10.1145/2535838.2535849.

13. A. Duret-Lutz. LTL translation improvements in Spot 1.0. *International Journal on Critical Computer-Based Systems*, 5(1/2):31–54, Mar. 2014. doi: 10.1504/IJCCBS.2014.059594.

14. A. Duret-Lutz, E. Renault, M. Colange, F. Renkin, A. G. Aisse, P. Schlehuber-Caissier, T. Medioni, A. Martin, J. Dubois, C. Gillard, and H. Lauko. From Spot 2.0 to Spot 2.10: What's new? In *Proceedings of the 34th International Conference on Computer Aided Verification (CAV'22)*, volume 13372 of *Lecture Notes in Computer Science*. Springer, Aug. 2022. doi: 10.1007/978-3-031-13188-2_9.

15. A. Duret-Lutz, S. Zhu, N. Piterman, G. De Giacomo, and M. Y. Vardi. Engineering an LTLf synthesis tool. In *Proceedings of the 29th International Conference on Implementation and Applications of Automata (CIAA'25)*, volume 15981 of *Lecture Notes in Computer Science*, pages 129–147. Springer, Sept. 2025. doi: 10.1007/978-3-032-02602-6_10.

16. C. Eisner and D. Fisman. *A Practical Introduction to PSL*. Series on Integrated Circuits and Systems. Springer, 2006. doi: 10.1007/978-0-387-36123-9.

17. B. Finkbeiner. Synthesis of reactive systems. In J. Esparza, O. Grumberg, and S. Sickert, editors, *Dependable Software Systems Engineering*, volume 45 of *NATO Science for Peace and Security Series - D: Information and Communication Security*, pages 72–98. IOS Press Ebooks, 2016. doi: 10.3233/978-1-61499-627-9-72.

18. M. J. Fischer and R. E. Ladner. Propositional dynamic logic of regular programs. *Journal of Computer and System Sciences*, 18(2):194–211, 1979. ISSN 0022-0000. doi: 10.1016/0022-0000(79)90046-1.

19. G. D. Giacomo and M. Y. Vardi. Linear temporal logic and linear dynamic logic on finite traces. In *Proceedings of the 33rd International Joint Conference on Artificial Intelligence (IJCAI'13)*, page 854–860. AAAI Press, 2013. ISBN 9781577356332. doi: 10.5555/2540128.2540252.

20. D. Giannakopoulou and F. Lerda. From states to transitions: Improving translation of LTL formulæ to Büchi automata. In D. Peled and M. Vardi, editors,

Proceedings of the 22nd IFIP WG 6.1 International Conference on Formal Techniques for Networked and Distributed Systems (FORTE'02), volume 2529 of *Lecture Notes in Computer Science*, Houston, Texas, Nov. 2002. Springer-Verlag. doi: 10.5555/646220.682186.

21. R. P. Kurshan. Complementing deterministic Büchi automata in polynomial time. *J. Comput. Syst. Sci.*, 35(1):59–71, Aug. 1987. doi: 10.1016/0022-0000(87)90036-5.

22. B. Le Saëc and I. Litovsky. On the minimization problem for ω-automata. In I. Prívara, B. Rovan, and P. Ruzicka, editors, *Proceedings of the 19th International Symposium on Mathematical Foundations of Computer Science (MFCS'94)*, volume 841 of *Lecture Notes in Computer Science*, pages 504–514, Kosice, Slovakia, Aug. 1994. Springer-Verlag. doi: 10.5555/645723.666714.

23. S. Lombardy and J. Sakarovitch. Two routes to automata minimization and the ways to reach it efficiently. In *Proceedings of the 23rd International Conference on Implementation and Application of Automata (CIAA'18)*, volume 10977 of *Lecture Notes in Computer Science*, pages 248–260. Springer, 2018. doi: 10.1007/978-3-319-94812-6_21.

24. A. Martin, E. Renault, and A. Duret-Lutz. Translation of semi-extended regular expressions using derivatives. In S. Z. Fazekas, editor, *Proceedings of the 28th International Conference on Implementation and Applications of Automata (CIAA'24)*, volume 15015 of *Lecture Notes in Computer Science*, pages 234–248. Springer Nature, Sept. 2024. doi: 10.1007/978-3-031-71112-1_17.

25. M. Michel. Algèbre de machines et logique temporelle. In M. Fontet and K. Mehlhorn, editors, *Proceedings of the Symposium on Theoretical Aspects of Computer Science (STACS'84)*, volume 166 of *Lecture Notes in Computer Science*, pages 287–298, Paris, Apr. 1984. doi: 10.1007/3-540-12920-0_26.

26. T. Varghese. *Parity and Generalized Büchi Automata — determinisation and complementation*. PhD thesis, University of Liverpool, Nov. 2014.

27. S. Xiao, J. Li, S. Zhu, Y. Shi, G. Pu, and M. Vardi. On-the-fly synthesis for LTL over finite traces. In *Proceedings of the 35th AAAI Conference on Artificial Intelligence (AAAI'21)*, volume 35, pages 6530–6537, May 2021. doi: 10.1609/aaai.v35i7.16809. Technical Tracks 7.