From C-Continuations to New Quadratic Algorithms For Automaton Synthesis

J.-M. Champarnaud D. Ziadi

LIFAR, Université de Rouen, 76821 Mont-Saint-Aignan, France
emails: {champarnaud, ziadi}@dir.univ-rouen.fr

Abstract

Two classical non-deterministic automata recognize the language denoted by a regular expression: the position automaton which deduces from the position sets defined by Glushkov and McNaughton-Yamada, and the equation automaton which can be computed via Mirkin's prebases or Antimirov's partial derivatives. Let $|E|$ be the size of the expression and $\|E\|$ be its alphabetic width, i.e. the number of symbol occurrences. The number of states in the equation automaton is less than or equal to the number of states in the position automaton, which is equal to $\|E\|+1$. On the other hand, the worst-case time complexity of Antimirov algorithm is $O(\|E\|^3 \cdot |E|^2)$, while it is only $O(\|E\| \cdot |E|)$ for the most efficient implementations yielding the position automaton (Brüggemann-Klein, Chang and Paige, Champarnaud *et al.*). We present an $O(|E|^2)$ space and time algorithm to compute the equation automaton. It is based on the notion of canonical derivative which makes it possible to efficiently handle sets of word derivatives. By the way, canonical derivatives also lead to a new $O(|E|^2)$ space and time algorithm to construct the position automaton.

1 Introduction

The notion of word derivative of a regular expression [7] and the related notions of continuation [6] and of partial derivative [3] are suitable tools to study the conversion problem of a regular expression into an equivalent finite automaton. Three fundamental results lead to the construction of three well-known automata. First, the set of the aci-dissimilar word derivatives of a regular expression is finite [7], which leads to the definition of

the deterministic derivative automaton. Secondly, the continuations w.r.t. a given symbol a in a linear expression (i.e. the non-null derivatives w.r.t. ua, for all the words u) are aci-similar [6], which yields a constructive interpretation of the position automaton (classically computed by Glushkov [12] and McNaughton-Yamada [15] algorithms). And third, the set of the partial derivatives of a regular expression is finite [3], which proves that the language denoted by the expression is recognized by the equation automaton [16, 3].

The notion of canonical derivative (c-derivative) developed by the authors in [9, 11] enlightens the tight connection between position automaton and equation automaton. In particular it is shown in [9] that (a) the c-continuations of a linear expression w.r.t. a symbol a (i.e. the non-null c-derivatives w.r.t. ua, for all the words u) graphically coincide, and (b) the set of the c-continuations of the linearized version of a regular expression reduces to the set of the partial derivatives of the expression by projection of positions on symbols. These theoretical results lead to the definition of the c-continuation automaton, whose main interest is the following: (1) it is isomorphic to the position automaton, and (2) it reduces to the equation automaton.

On the other hand, these theoretical results generate new algorithmic developments. Let us recall first some features of previous automaton constructions based on derivatives. Their space and time complexity essentially depends on the size of derivatives, and on the way aci-similarity tests are achieved. There are two main pitfalls: (1) the size of word derivatives, which may be exponential w.r.t. the size of the expression, and (2) the cost of direct aci-similarity tests, since they involve expensive string manipulations. These two features make it clever to implement Brzozowski's algorithm [2]. Similarly, an explicit computation of the set of continuations would lead to an inefficient implementation of Berry-Sethi's algorithm [6], due to the possibly exponential size of continuations; the number of aci-similarity tests can be reduced however, since two continuations w.r.t. the same symbol are known to be aci-similar. Conversely, partial derivatives have a reasonable quadratic size but aci-similarity tests remain time consuming: the construction of the equation automaton [3] is in $O(\|E\|\cdot|E|^2)$ space and $O(\|E\|^3\cdot|E|^2)$ time, where $|E|$ is the size of the expression and $\|E\|$ its alphabetic width, i.e. the number of symbol occurrences.

Let us point out now that the challenge is to compute the equation automaton, which is smaller than the positions one, with the same complexity as the most efficient implementations yielding the position automaton. The notion of c-derivative makes it possible: c-derivatives being related both

to continuations and to partial derivatives, they allow to deduce the set of partial derivatives of a regular expression from the computation of the c-continuations in its linearized version. From an algorithmic point of view, the two pitfalls we have mentioned are bypassed thanks to the following properties: (1) c-derivatives have a quadratic size, thus an explicit computation of the set of c-continuations can be performed, (2) identification of c-continuations into partial derivatives can be achieved via efficient string procedures, and (3) final states and transitions can be produced by procedures based on the specific structure of the c-derivatives and on their connection to positions. As a result, efficient implementations of the NFA constructions can be designed, in particular $O(|E|^2)$ worst case time and space implementations of both the equation automaton and the position automaton.

Section 2 recalls some useful notations, definitions and classical results of automata theory, in particular about the notions of word derivatives, continuations and partial derivatives.

Section 3 summarizes theoretical results concerning c-derivatives and c-continuations of a regular expression, and their relations with word derivatives and partial derivatives. The definition of the c-continuation automaton is recalled, as well as the way it is connected to the position automaton and to the equation automaton.

Section 4 gives an algorithmic analysis of the $O(\|E\| \cdot |E|^2)$ time and space construction of the c-continuation automaton, based on an explicit computation of the c-continuations.

Section 5 details the algorithmic refinements leading to an $O(|E|^2)$ space and time construction of both the position automaton and the equation automaton.

2 Preliminaries

We first recall some definitions, and basic properties concerning regular languages, regular expressions, finite automata and derivatives. For further details about these topics, we refer to classical books [4, 13] or handbooks [19].

2.1 Regular expressions and languages

Let Σ be a non-empty finite set of symbols, called the alphabet. Σ^* represents the set of all the words over Σ. The empty word is denoted by ε. A

language over Σ is a subset of Σ^*.

Regular expressions over an alphabet Σ and *regular languages* they denote are inductively defined as follows:

(1) 0, 1, and a, for $a \in \Sigma$, are regular expressions; 0 denotes the language $L(0) = \emptyset$, 1 denotes the language $L(1) = \{\varepsilon\}$, and a, for $a \in \Sigma$, denotes the language $L(a) = \{a\}$.

(2) Let F (resp. G) be a regular expression denoting the language $L(F)$ (resp. L(G)). Then $F + G$, $F \cdot G$, also written FG, and F^* are regular expressions. $F + G$ denotes the language $L(F + G) = L(F) \cup L(G)$, $F \cdot G$ denotes the language $L(F \cdot G) = L(F)L(G)$ and F^* denotes the language $L(F^*) = L(F)^*$.

A regular expression over an alphabet Σ can be viewed as a term on the set $\Sigma \cup \{0, 1, *, +, \cdot\}$. Properties of $*$, $+$ and $\cdot$ operations and of 0 and 1 constants lead to identification rules such as: $0 + E = E = E + 0$, $1 \cdot E = E = E \cdot 1$, $0 \cdot E = 0 = E \cdot 0$. Associativity, commutativity and idempotence of the $+$ operation lead to the so-called aci-rules: two regular expressions F and G are said to be aci-similar ($F \sim_{aci} G$) if and only if they reduce to the same expression by applying aci-rules. We shall write: $F \equiv G$ if two regular expressions graphically coincide, i.e. are the same string of symbols.

Let E be a regular expression over the alphabet Σ. The *alphabetic width* of E, denoted $\|E\|$, is the number of occurrences of symbols in E. E is said to be *linear* over Σ if and only if every symbol of Σ occurs (at most) one time in E. For all j in $[1, \|E\|]$, if x is the j^{th} occurrence of symbol in E, the pair (x, j) is called a *position* of E. By abuse of notation (x, j) is written x_j. The set of positions of E is denoted by Pos_E: $Pos_E = \{x_j | x \in \Sigma, j \in [1, \|E\|]\}$. The expression $\overline{E}$ over Pos_E deduced from E by replacing symbol x in place j by x_j, for all j in $[1, \|E\|]$, is called *the linearized version* of E. $\overline{E}$ is linear over Pos_E. We denote h the mapping from Pos_E to Σ such that $h(x_i) = x$, $\forall i \in [1, \|E\|]$, and $h(\overline{E}) \equiv E$. Let $E = a \cdot (a + b) + (a + b) \cdot (1 + b)$. We have: $Pos_E = \{a_1, a_2, b_3, a_4, b_5, b_6\}$, $\overline{E} = a_1 \cdot (a_2 + b_3) + (a_4 + b_5) \cdot (1 + b_6)$, $h(a_1) = h(a_2) = h(a_4) = a$ and $h(b_3) = h(b_5) = h(b_6) = b$.

Let E be a regular expression. Let $\lambda(E) = 1$ if $\varepsilon \in L(E)$ and 0 otherwise. E is said to be *nullable* if and only if $\lambda(E) = 1$.

2.2 Finite automata and recognizable languages

Let Σ be a finite alphabet. A *finite automaton* over Σ is a 5-tuple $\mathcal{M} = (Q, \Sigma, I, T, E)$ where Q is a set of *states*, I a subset of Q whose elements are

the *initial states*, T a subset of Q whose elements are the *final states*, E a subset of the Cartesian product $Q \times \Sigma \times Q$ whose elements are the *edges*. The automaton $\mathcal{M}$ is *deterministic* if there is only one initial state and if for all $(q, a) \in Q \times \Sigma$ there is at most one state q' such that $(q, a, q') \in E$. $\mathcal{M}$ is said to be a *DFA* if it is deterministic and a *NFA* otherwise. We shall write $\mathcal{M} = (Q, \Sigma, i, T, E)$ for an automaton with a unique initial state i. Edges are also called *transitions* and can be represented by a transition function δ from $Q \times \Sigma$ to Q when $\mathcal{M}$ is a DFA, and from $Q \times \Sigma$ to 2^Q when $\mathcal{M}$ is a NFA.

A *path* of $\mathcal{M}$ is a sequence (q_i, a_i, q_{i+1}), $i = 1, \ldots, n$, of consecutive edges. Its *label* is the word $w = a_1 a_2 \ldots a_n$. A word $w = a_1 a_2 \ldots a_n$ is *recognized* by the automaton $\mathcal{M}$ if there is a path with label w such that $q_1 \in I$ and $q_{n+1} \in T$. The language *recognized* by the automaton $\mathcal{M}$ is the set of words which it recognizes. A language L is *recognizable* if and only if there exists a finite automaton whose language is L. Kleene theorem [14] states that the set $Rat(\Sigma^*)$ of regular languages over Σ and the set $Reg(\Sigma^*)$ of recognizable languages over Σ are equal.

2.3 The position automaton of a regular expression

Let E be a linear expression over Σ. We consider the following sets of symbols:

- $First(E)$, the set of symbols that match the first symbol of some word in $L(E)$.
- $Last(E)$, the set of symbols that match the last symbol of some word in $L(E)$.
- $Follow(E, x)$, for all x in Σ: the set of symbols that follow the symbol x in some word of $L(E)$.

If E is a regular expression over Σ, we consider its linearized version $\overline{E}$, whose symbols are the positions of E, and we set: $First(E) = First(\overline{E})$, $Last(E) = Last(\overline{E})$, $Follow(E, x) = Follow(\overline{E}, x)$, for all x in Pos_E.

Let h be the mapping from Pos_E to Σ induced by the linearization of E over Pos_E. $\mathcal{P}_E$, the position automaton of E, whose states are the positions of E, and which recognizes $L(E)$, is defined as follows.

Definition 1 (position automaton) *The position automaton of* E, $\mathcal{P}_E = (Q, \Sigma, i, T, \delta)$, *is defined by:* $Q = Pos_E \cup \{0\}$, $i = \{0\}$, $T = [$*if* $\lambda(E) = 0$ *then*

Last(E) *else* *Last*$(E) \cup \{0\}]$, $\delta(0,a) = \{x \in First(E) \mid h(x) = a\}$, $\forall a \in \Sigma$, *and* $\delta(x,a) = \{y \mid y \in Follow(E,x)$ *and* $h(y) = a\}$, $\forall x \in Pos_E$ *and* $\forall a \in \Sigma$.

The automaton $\mathcal{P}_E$ can be built in $O(\|E\| \cdot |E|^2)$ time by Glushkov [12] and McNaughton-Yamada [15] algorithms, or in $O(\|E\| \cdot |E|)$ time by optimized implementations [5, 8, 20, 18].

Example 1 *Consider the regular expression* $E = ((x^*y)^* + x(x^*y)^*y)^*$. *The linearized version of* E *is* $\overline{E} = ((x_1^*y_2)^* + x_3(x_4^*y_5)^*y_6)^*$. *We have:* $First(E) = \{x_1, y_2, x_3\}$, $Last(E) = \{y_2, y_6\}$, $\lambda(E) = 1$ *and* $Follow(E, x_1) = \{x_1, y_2\}$, $Follow(E, y_2) = \{x_1, y_2, x_3\}$, $Follow(E, x_3) = Follow(E, y_5) = \{x_4, y_5, y_6\}$, $Follow(E, x_4) = \{x_4, y_5\}$, $Follow(E, y_6) = \{x_1, y_2, x_3\}$.

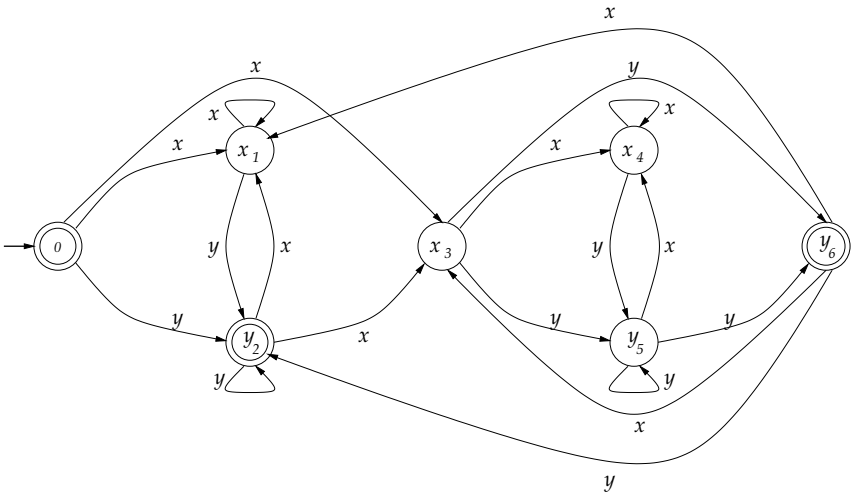

Figure 1: The position automaton for $E = ((x^*y)^* + x(x^*y)y)^*$.

2.4 Derivatives

2.4.1 Word derivatives

Word derivatives of regular expressions have been introduced by Brzozowski in [7].

Definition 2 (word derivative) *Let* E *be a regular expression,* a *be a symbol in* Σ *and* $u = u_1 \dots u_n$ *be a word in* Σ^*. *The derivative* $u^{-1}E$ *of* E *w.r.t.* u *is recursively defined as follows:*

$$a^{-1}0 \;=\; 0$$

$$
\begin{array}{rcl}
a^{-1}1 & = & 0 \\
a^{-1}x & = & 1 \quad \textit{if } a = x \quad 0 \textit{ otherwise} \\
a^{-1}(F+G) & = & a^{-1}F + a^{-1}G \\
a^{-1}(F \cdot G) & = & \begin{cases} a^{-1}F \cdot G & \textit{if } \lambda(F) = 0 \\ a^{-1}F \cdot G + a^{-1}G & \textit{otherwise} \end{cases} \\
a^{-1}(F^*) & = & a^{-1}F \cdot F^* \\
\varepsilon^{-1}E & = & E \\
(u_1 \ldots u_n)^{-1}E & = & (u_2 \ldots u_n)^{-1}({u_1}^{-1}E)
\end{array}
$$

2.4.2 Partial derivatives

The notion of partial derivative of a regular expression has been introduced by Antimirov in [3]. As noticed in [10], it is very close to the notion of prebase due to Mirkin [16]. It leads to a nondeterministic factorization of the expression.

Definition 3 (set of partial derivatives w.r.t. a symbol) *Given a regular expression E and a symbol a, the set of partial derivatives of E w.r.t. a, written $\partial_a(E)$, is recursively defined on the structure of E as follows:*

$$
\begin{array}{rcl}
\partial_a(0) & = & \emptyset \\
\partial_a(1) & = & \emptyset \\
\partial_a(x) & = & \{1\} \quad \textit{if } a = x \quad \emptyset \textit{ otherwise} \\
\partial_a(F+G) & = & \partial_a(F) \cup \partial_a(G) \\
\partial_a(F \cdot G) & = & \begin{cases} \emptyset & \textit{if } G = 0 \\ \partial_a(F) \cdot G & \textit{if } G \neq 0 \textit{ and } \lambda(F) = 0 \\ \partial_a(F) \cdot G \cup \partial_a(G) & \textit{otherwise} \end{cases} \\
\partial_a(F^*) & = & \partial_a(F) \cdot F^*
\end{array}
$$

Example 2 *Let $E = ((x^*y)^* + x(x^*y)^*y)^*$. We have:*

$$
\begin{array}{rcl}
\partial_x(E) & = & \partial_x((x^*y)^* + x(x^*y)^*y)((x^*y)^* + x(x^*y)^*y)^* \\
& = & (\partial_x((x^*y)^*) \cup \partial_x(x(x^*y)^*y))((x^*y)^* + x(x^*y)^*y)^* \\
& = & (\partial_x(x^*y)(x^*y)^* \cup \partial_x(x)(x^*y)^*y)((x^*y)^* + x(x^*y)^*y)^* \\
& = & (\partial_x(x)x^*y(x^*y)^* \cup \{(x^*y)^*y\})((x^*y)^* + x(x^*y)^*y)^* \\
& = & \{x^*y(x^*y)^*((x^*y)^* + x(x^*y)^*y)^* \ , \ (x^*y)^*y((x^*y)^* + x(x^*y)^*y)^*\}
\end{array}
$$

Definition 4 (extensions) *The symbol a in $\partial_a(E)$ can be replaced by any word u of Σ^* or by any set of words U, and the expression E can be replaced by any set R of expressions, according to the formulas:*

$$\begin{aligned}
\partial_\varepsilon(E) &= \{E\} \\
\partial_{ua}(E) &= \partial_a(\partial_u(E)) \\
\partial_u(R) &= \bigcup_{E\in R} \partial_u(E) \\
\partial_U(E) &= \bigcup_{u\in U} \partial_u(E)
\end{aligned}$$

Let $\mathcal{PD}(E) = \partial_{\Sigma^*}(E)$ be the set of all partial derivatives of the regular expression E.

Theorem 1 (Antimirov [3]) *The cardinality of the set $\mathcal{PD}(E)$ of all partial derivatives of a regular expression E is less than or equal to $\|E\| + 1$.*

Hence the definition of $\mathcal{E}_E$, the equation automaton of E, whose states are the partial derivatives of E, and which recognizes $L(E)$.

Definition 5 (equation automaton) *The equation automaton of a regular expression E, $\mathcal{E}_E = (Q, \Sigma, i, T, \delta)$, is defined by: $Q = \mathcal{PD}(E)$, $i = E$, $T = \{p \mid \lambda(p) = 1\}$ and $\delta(p, a) = \partial_a(p)$, $\forall p \in Q$ and $\forall a \in \Sigma$.*

Example 3 *Let $E = ((x^*y)^* + x(x^*y)y)^*$. The set of states $Q = \mathcal{PD}(E)$ of the equation automaton is: $\{p_0 = ((x^*y)^*+x(x^*y)^*y)^*$, $p_1 = (x^*y)^*y((x^*y)^*+ x(x^*y)^*y)^*\}$, $p_2 = (x^*y)^*((x^*y)^* + x(x^*y)^*y)^*$, $p_3 = x^*y(x^*y)^*y((x^*y)^* + x(x^*y)^*y)^*$, $p_4 = x^*y(x^*y)^*((x^*y)^* + x(x^*y)^*y)^*\}$.*

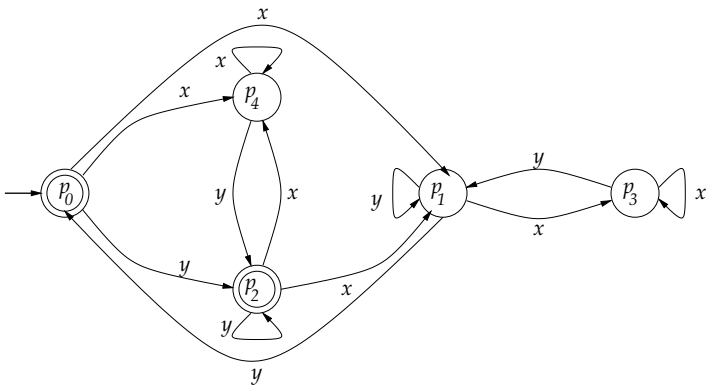

Figure 2: The equation automaton for $E = ((x^*y)^* + x(x^*y)y)^*$.

3 Canonical derivatives and continuations

Our new algorithms are based on the theoretical background we introduced in [9, 11]. We review the notion of c-derivative of a regular expression, its relationship to word and partial derivatives, the notion of c-continuation of a linear expression, the definition of the c-continuation automaton, as well as the way this automaton is connected to the positions one and to the equations one. Proofs of propositions and theorems can be found in [9].

3.1 Canonical derivatives

Definition 6 (c-derivative) *Let E be a regular expression, a be a symbol in Σ and $u = u_1 \dots u_n$ be a word in Σ^*. The c-derivative $d_u(E)$ of E w.r.t. u is recursively defined as follows:*

$$
\begin{aligned}
d_a(0) &= 0 \\
d_a(1) &= 0 \\
d_a(x) &= \begin{cases} 1 & \text{if } a = x \\ 0 & \text{otherwise} \end{cases} \\
d_a(F+G) &= \begin{cases} d_a(F) & \text{if } d_a(F) \neq 0 \\ d_a(G) & \text{otherwise} \end{cases} \\
d_a(F \cdot G) &= \begin{cases} d_a(F) \cdot G & \text{if } d_a(F) \neq 0 \\ \lambda(F) \cdot d_a(G) & \text{otherwise} \end{cases} \\
d_a(F^*) &= d_a(F) \cdot F^* \\
d_\varepsilon(E) &= E \\
d_{u_1 \dots u_n}(E) &= d_{u_2 \dots u_n}(d_{u_1}(E))
\end{aligned}
$$

Proposition 1 *The c-derivative $d_u(E)$ of a linear expression E w.r.t. a word u of Σ^+ is either 0 or such that:*

$$
\begin{aligned}
d_u(u) &= 1 \\
d_u(F+G) &= \begin{cases} d_u(F) & \text{if } d_u(F) \neq 0 \\ d_u(G) & \text{otherwise} \end{cases} \\
d_u(F \cdot G) &= \begin{cases} d_u(F) \cdot G & \text{if } d_u(F) \neq 0 \\ d_s(G) & \text{otherwise } (s \neq \varepsilon \text{ is some suffix of } u) \end{cases} \\
d_u(F^*) &= d_s(F) \cdot F^* \quad (s \neq \varepsilon \text{ is some suffix of } u)
\end{aligned}
$$

Example 4 *Consider the linear expression* $\overline{E} = (x_1^* y_2)^* + x_3(x_4^* y_5)^* y_6$. *We have:*

$$\begin{aligned} d_{x_1}(\overline{E}) &= d_{x_1}((x_1^* y_2)^* + x_3(x_4^* y_5)^* y_6) \\ &= d_{x_1}((x_1^* y_2)^*) = d_{x_1}(x_1^* y_2)(x_1^* y_2)^* \\ &= d_{x_1}(x_1) x_1^* y_2 (x_1^* y_2)^* = x_1^* y_2 (x_1^* y_2)^* \end{aligned}$$

Next proposition and next theorem enlighten the relations between c-derivatives and word derivatives.

Proposition 2 *Let* E *be a linear expression. For all* $a \in \Sigma$, *and for all* $u \in \Sigma^*$, *one has:*

$$d_a(E) \equiv a^{-1}E \text{ and } d_u(E) \sim_{aci} u^{-1}E$$

Example 5 *Consider the linear expression* $E = (a^* + b)^*$. *In this example we can see that* $d_a(E) = a^{-1}E$ *and that* $d_{aa}(E) \sim_{aci} (aa)^{-1}E$.
$d_a((a^* + b)^*) = a^*(a^* + b)^* = a^{-1}(a^* + b)^*$,
$d_{aa}((a^* + b)^*) = a^*(a^* + b)^*$,
$(aa)^{-1}(a^* + b)^* = a^*(a^* + b)^* + a^*(a^* + b)^*$.

Theorem 2 *Let* E *be a linear expression over an alphabet* Σ. *For all* $u, v \in \Sigma^*$, *one has:*

$$u^{-1}E \sim_{aci} v^{-1}E \Rightarrow d_u(E) \equiv d_v(E)$$

The following theorem is close to Berry-Sethi's one [6] concerning continuations in a linear expression.

Theorem 3 *If* E *is linear, for any symbol* a *and any word* u, *the c-derivative* $d_{ua}(E)$ *of* E *w.r.t.* ua *is either* 0 *or unique.*

Theorem 3 allows us to define $c_a(E)$, the c-continuation of a in E, which is the unique value of the non-null c-derivatives $d_{ua}(E)$. From Proposition 1, we deduce the following formulas.

Proposition 3 *For any symbol a of a linear expression E, the c-continuation $c_a(E)$ is such that:*

$$\begin{aligned} c_a(a) &= 1 \\ c_a(F+G) &= \begin{cases} c_a(F) & \text{if } c_a(F) \neq 0 \\ c_a(G) & \text{otherwise} \end{cases} \\ c_a(F\cdot G) &= \begin{cases} c_a(F)\cdot G & \text{if } c_a(F) \neq 0 \\ c_a(G) & \text{otherwise} \end{cases} \\ c_a(F^*) &= c_a(F)\cdot F^* \end{aligned}$$

Proposition 3 allows us to compute canonical representatives for Berry-Sethi sets of continuations.

Corollary 1 *For any symbol a of a linear expression E, the c-continuation $c_a(E)$ is either 1 or a subexpression of E or a product of subexpressions.*

More precisely, $c_a(E) = H_1 \cdot H_2 \cdot \ldots H_l$, where H_i is a subexpression of E, for all $1 \leq i \leq l$. This decomposition is fundamental for the computation of c-continuations on the syntax tree of the expression, as it will be seen in the next section.

Example 6 *Let $\overline{E} = (x_1^* y_2)^* + x_3(x_4^* y_5)^* y_6$. We have: $c_{x_1}(\overline{E}) = x_1^* \cdot y_2 \cdot (x_1^* y_2)^*$.*

3.2 The c-continuation automaton

Let E be a regular expression over Σ, $\overline{E}$ be the linearized version of E over Pos_E and h be the mapping from Pos_E to Σ. We assume 0 is a symbol not in Pos_E. Let $c_0 = d_\varepsilon(\overline{E}) = \overline{E}$ and c_x be an abbreviation of $c_x(\overline{E})$.

Theorem 3 leads to the definition of the non-deterministic automaton $\mathcal{C}_E$, called the c-continuation automaton of E. States are pairs (x, c_x) with x in $Pos_E \cup \{0\}$. Transitions are deduced from the computation of c-continuations c_x.

Definition 7 (c-continuation automaton) *The c-continuation automaton of E, $\mathcal{C}_E = (Q, \Sigma, i, T, \delta)$, is defined by: $Q = \{(x, c_x) | x \in Pos_E \cup \{0\}\}$, $i = (0, c_0)$, $T = \{(x, c_x) \mid \lambda(c_x) = 1\}$, $\delta((x, c_x), a) = \{(y, c_y) \mid h(y) = a \text{ and } d_y(c_x) \equiv c_y\}$, $\forall x \in Pos_E \cup \{0\}$ and $\forall a \in \Sigma$.*

Example 7 *Let* $E = ((x^*y)^* + x(x^*y)y)^*$ *and* $\overline{E}$ *be its linearized version. The states of the c-continuation automaton are:*
$(0, c_0) = (0, ((x_1^*y_2)^* + x_3(x_4^*y_5)^*y_6)^*)$,
$(x_1, c_{x_1}) = (x_1,\ x_1^*y_2(x_1^*y_2)^*((x_1^*y_2)^* + x_3(x_4^*y_5)^*y_6)^*)$,
$(y_2, c_{y_2}) = (y_2,\ (x_1^*y_2)^*((x_1^*y_2)^* + x_3(x_4^*y_5)^*y_6)^*)$,
$(x_3, c_{x_3}) = (x_3,\ (x_4^*y_5)^*y_6((x_1^*y_2)^* + x_3(x_4^*y_5)^*y_6)^*)$,
$(x_4, c_{x_4}) = (x_4,\ x_4^*y_5(x_4^*y_5)^*y_6((x_1^*y_2)^* + x_3(x_4^*y_5)^*y_6)^*$,
$(y_5, c_{y_5}) = (y_5,\ (x_4^*y_5)^*y_6((x_1^*y_2)^* + x_3(x_4^*y_5)^*y_6)^*))$,
$(y_6, c_{y_6}) = (y_6,\ ((x_1^*y_2)^* + x_3(x_4^*y_5)^*y_6)^*)$.

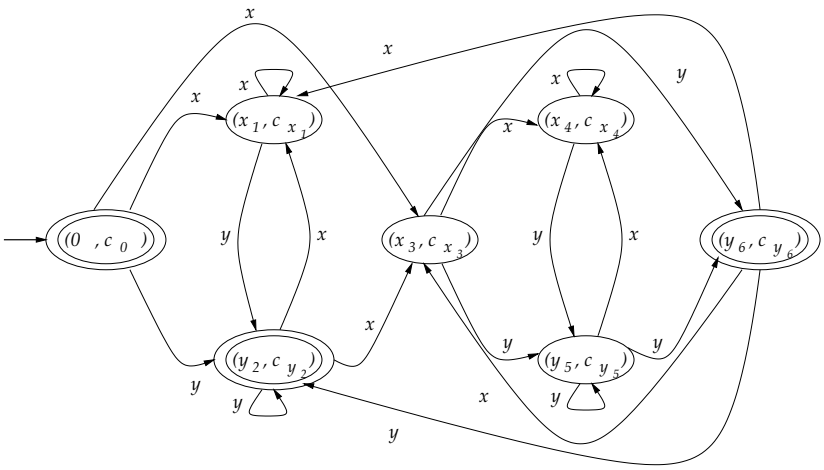

Figure 3: The c-continuation automaton for $E = ((x^*y)^* + x(x^*y)y)^*$.

The $\mathcal{C}_E$ automaton coincides with the position automaton of E if its states are viewed as positions, and it reduces to the equation automaton if its states are viewed as c-continuations.

Let $p : Q \to Pos_E \cup \{0\}$ such that $p(x, c_x) = x$. The automaton $p(\mathcal{C}_E)$ deduces from $\mathcal{C}_E$ by replacing Q by $Pos_E \cup \{0\}$.

Theorem 4 *Let* E *be a regular expression. The automaton* $p(\mathcal{C}_E)$*, based on the c-continuations of* $\overline{E}$*, and* $\mathcal{P}_E$*, the position automaton, are identical.*

The proof deduces from the following proposition:

Proposition 4 *Let* E *be a regular expression. The following equalities hold:*

1. $First(E) = \{y \in Pos_E \mid d_y(\overline{E}) \neq 0\}$;

2. $Last(E) = \{y \in Pos_E \mid \lambda(c_y(\overline{E})) = 1\}$;

3. $Follow(E, x) = \{y \in Pos_E \mid d_y(c_x(\overline{E})) \neq 0\}$.

In order to explain how $\mathcal{C}_E$ reduces to $\mathcal{E}_E$, the equation automaton of E, we first recall the relation between c-derivatives and partial derivatives.

The following proposition says that the set of partial derivatives of a regular expression E w.r.t. a symbol a is equal to the set of the projections on Σ of the c-continuations in $\overline{E}$ w.r.t. positions a_i such that $h(a_i) = a$.

Proposition 5 *Let E be a regular expression. Let H_E be a subexpression of E. Denote by $P(H_E)$ the property: $a_i \in Pos_E(H_E) \wedge h(a_i) = a \wedge d_{a_i}(\overline{H_E}) \neq 0$. Then for all H_E one has:*

$$\bigcup_{P(H_E)} h(d_{a_i}(\overline{H_E})) = \partial_a(H_E)$$

The c-derivative of a linear expression w.r.t. a position is generally not linear. Let $E = (ab+b)^*$. The expression $F = d_{a_1}(\overline{E}) = b_2(a_1b_2+b_3)^*$ is not linear. Let $G = \overline{h(F)} = \overline{b(ab+b)^*} = b_1(a_2b_3+b_4)^*$ be the linearized version of $h(F)$ and h' the associated mapping from $Pos_{h(F)}$ to Σ (notice that the sets of indices in F and G are independent). The structure of c-derivatives is such that the c-derivative of F w.r.t. b_2 and the c-derivatives of $G = \overline{h(F)}$ w.r.t. b_1 and b_3 have identical projections on Σ: $h(d_{b_2}(F)) = h'(d_{b_1}(G)) = h'(d_{b_3}(G))$.

It is the reason why Proposition 5 extends to derivations w.r.t. a word, as stated by Theorem 5.

Theorem 5 *Let E be a regular expression. Let H_E be a subexpression of E. $P(u, H_E)$ denotes the property: $v = \overline{u} \in Pos_E^*(H_E) \wedge h(v) = u \wedge d_v(\overline{H_E}) \neq 0$. Then for all H_E, one has:*

$$\bigcup_{P(H_E,u)} h(d_{\overline{u}}(\overline{H_E})) = \partial_u(H_E)$$

Example 8 *Let $E = (x^*y)^* + x(x^*y)^*y$ and $\overline{E} = (x_1^*y_2)^* + x_3(x_4^*y_5)^*y_6$ be its linearized version. We have:*
$d_{x_1x_1}(\overline{E}) = x_1^*y_2(x_1^*y_2)^*$, $d_{x_3x_4}(\overline{E}) = x_4^*y_5(x_4^*y_5)^*y_6$,
and $d_{x_1x_3}(\overline{E}) = d_{x_1x_4}(\overline{E}) = d_{x_3x_1}(\overline{E}) = d_{x_3x_3}(\overline{E}) = d_{x_4x_1}(\overline{E}) = d_{x_4x_3}(\overline{E}) = d_{x_4x_4}(\overline{E}) = 0$.
Hence $\partial_{xx}(E) = \{x^*y(x^*y)^* \ , \ x^*y(x^*y)^*y\}$.

Let $\sim$ be the equivalence relation on the set of states of $\mathcal{C}_E$ defined by:

$$(x, c_x) \sim (y, c_y) \Leftrightarrow h(c_x) \equiv h(c_y)$$

Let us denote by $[c_x]$ the equivalence class of the state (x, c_x). The relation $\sim$ is proved to be right-invariant, i.e. for all a in Σ, for all (x, c_x), (y, c_y) in Q such that $(x, c_x) \sim (y, c_y)$, we have: $\delta((x, c_x), a))/_\sim = \delta((y, c_y), a))/_\sim$. Hence the definition of the quotient automaton of $\mathcal{C}_E$ modulo the relation $\sim$.

Definition 8 (quotient automaton) *The automaton* $\mathcal{C}_E/_\sim = (Q_\sim, \Sigma, i, T, \delta)$ *is defined as follows:* $Q_\sim = \{[c_x] \mid x \in Pos_E \cup \{0\}\}$, $i = [c_0]$, $T = \{[c_x] \mid \lambda(c_x) = 1\}$, *and* $\delta([c_x], a) = \{[c_y] \mid h(y) = a$ *and* $d_y(c_x) \equiv c_y\}$, $\forall [c_x] \in Q_\sim$ *and* $\forall a \in \Sigma$.

Theorem 6 *Let E be a regular expression. The quotient automaton $\mathcal{C}_E/_\sim$ is isomorphic to $\mathcal{E}_E$, the equation automaton of E.*

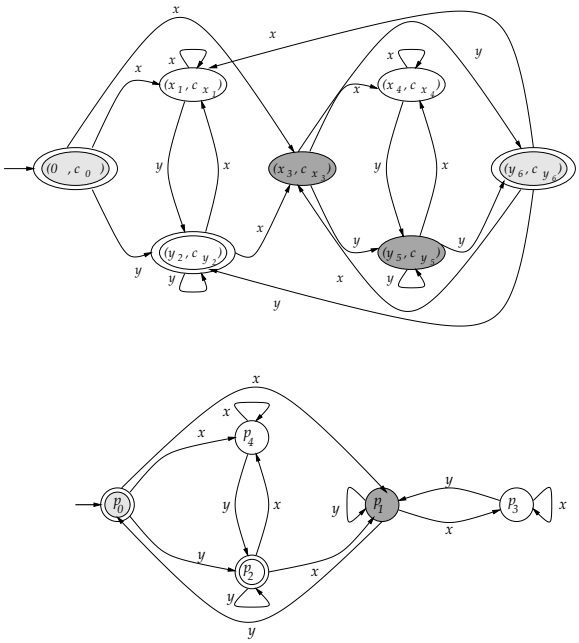

Figure 4: $\mathcal{C}_E$ and $\mathcal{C}_E/_\sim$ for $E = ((x^*y)^* + x(x^*y)^*y)^*$.

Theorems 4 and 6 deepen relations between position automaton and equation automaton through c-continuation automaton. Furthermore, they originate significant improvements of NFA constructions.

4 Cubic constructions based on c-continuations

The design of the new algorithms we present in this paper are based on properties and procedures which are now detailed. Since c-continuations in a linear expression E have a quadratic size w.r.t. the size of the expression, any NFA construction involving an explicit computation of the list of c-continuations has at least an $O(\|E\| \cdot |E|^2)$ space and time complexity. It is the case of the two first algorithms we describe: *AlgoC* to compute the automaton $\mathcal{C}_E$ and *AlgoCQuotient* to compute $\mathcal{C}_E/_\sim$. Since the complexity of the whole construction is cubic, we first provide elementary versions for procedures yielding the set of final states and the set of transitions. Improvements will be presented in the next section, leading to a quadratic construction of the automata $\mathcal{P}_E$ and $\mathcal{E}_E$. Let us point out that these procedures and improvements make use of the connection between c-continuations and position sets; thus they are related to the procedures used in ZPC-algorithm [20, 18]. However they are original since computation is deduced from the structure of the c-continuations.

From now on, we use abbreviations s for $|E|$ and w for $\|E\|$. Notice that the alphabetic width and the size of a regular expression are a priori independent parameters, since the size of an expression can be arbitrarily increased by adding 0s, concatenating 1s, or starring star expressions, with a constant alphabetic width (and with the same recognized language). Therefore complexities are expressed w.r.t. both of these two parameters.

Most of the computations are processed over the syntax tree $T(E)$ of E which is built in $O(s)$ space and time. Notice that $|E| = |\overline{E}|$ and $\|E\| = \|\overline{E}\|$. Notice too that $T(E)$ and $T(\overline{E})$ only differ by the labels of their leaves.

The sketch is similar in *AlgoC* and *AlgoCQuotient*:

Part 1:	Compute the set of states.
Part 2:	Compute the set of final states.
Part 3:	Compute the set of transitions.

Part 1 of both algorithms begins with the explicit computation of the c-

continuations. *AlgoCQuotient* includes an additional step to identify equivalent c-continuations.

4.1 C-continuations: computation

We first examine the complexity of c-continuations computation. Notice that computations generally involve w c-continuations added with the expression E itself. It is the reason why we speak of $w+1$ c-continuations.

By Corollary 1, the c-continuation $c_a(E)$ of the symbol a in the linear expression E is a product of distinct subexpressions H_i of E, possibly reduced to a single subexpression or to 1. The following proposition shows how this product can be computed over the syntax tree.

Proposition 6 *Let F and G be subexpressions of the linear expression E. Let f be the mapping such that:*

$$f(F) = \begin{cases} F^* & \text{if } F \text{ is a son of } F^* \text{ in } T_E \\ G & \text{if } F \text{ is a son of } F \cdot G \text{ in } T_E \\ 1 & \text{otherwise} \end{cases}$$

Let $\bigodot$ denote the concatenation of a list of expressions. We write $F \prec G$ if G is an ancestor of F. Then for all symbol a in Σ_E, we have:

$$c_a(E) \;=\; \bigodot_{a \preceq H \prec E} f(H)$$

Before giving a proof, we illustrate this proposition by an example.

Example 9 *Let $E = ((x^*y)^* + x(x^*y)y)^*$. In Figure 5, we can see that:*

$$\begin{aligned} c_{x_4}(\overline{E}) &= f(x_4) \cdot f(x_4^*) \cdot f(x_4^*y_5) \cdot f((x_4^*y_5)^*) \cdot f((x_1^*y_2)^* + x_3(x_4^*y_5)^*y_6) \\ &= x_4^* \cdot y_5 \cdot (x_4^*y_5)^* \cdot y_6 \cdot ((x_1^*y_2)^* + x_3(x_4^*y_5)^*y_6)^* \end{aligned}$$

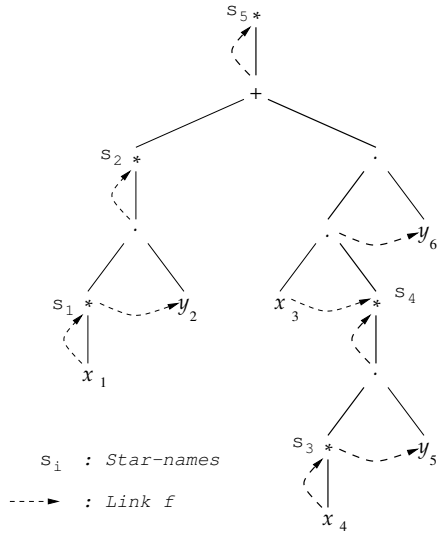

Figure 5: Links f for $E = ((x^*y)^* + x(x^*y)y)^*$.

Proof. The proof is by induction on the number of operators in E. Let a and x be symbols in Σ. We have $d_x(0) = d_x(1) = 0$ for all $x \in \Sigma$ and $d_x(a) = 0$ for all $x \in \Sigma$, $x \neq a$. On the other hand, $c_a(a) = 1$ and $\bigodot_{a \preceq F \prec a} f(F) = f(a) = 1$. So the theorem is true for the base case. Assume now that it is true for all expressions with less than n, $n \geq 1$, operators and let us consider an expression E with n operators. E has one of the three forms (a) $E = F + G$, (b) $E = F \cdot G$ or (c) $E = F^*$.

Case a: $E = F + G$. If $a \in \Sigma_F$, then by induction hypothesis we have:

$$c_a(F) \quad = \quad \bigodot_{a \preceq H \prec F} f(H)$$

On the other hand, we have $c_a(E) = c_a(F)$ and $f(F) = 1$, so we can write:

$$c_a(F) \quad = \quad (\bigodot_{a \preceq H \prec F} f(H)) \cdot f(F)$$

Finally we get:

$$c_a(E) \quad = \quad \bigodot_{a \preceq H \prec E} f(H)$$

A similar proof can be done when $a \in \Sigma_G$.

Case b: $E = F \cdot G$. If $a \in \Sigma_F$, then we have: $c_a(E) = c_a(F) \cdot G$.

$$\begin{aligned} c_a(E) &= (\bigodot_{a \preceq H \prec F} f(H)) \cdot G, \text{ by induction hypothesis} \\ &= (\bigodot_{a \preceq H \prec F} f(H)) \cdot f(F), \text{ since } f(F) = G \\ &= \bigodot_{a \preceq H \prec E} f(H) \end{aligned}$$

If $a \in \Sigma_G$, then we have: $c_a(E) = c_a(G)$.

$$\begin{aligned} c_a(E) &= \bigodot_{a \preceq H \prec G} f(H), \text{ by induction hypothesis} \\ &= (\bigodot_{a \preceq H \prec G} f(H)) \cdot f(G), \text{ since } f(G) = 1 \\ &= \bigodot_{a \preceq H \prec E} f(H) \end{aligned}$$

Case c: $E = F^*$. We have: $c_a(E) = c_a(F) \cdot F^*$.

$$\begin{aligned} c_a(E) &= \bigodot_{a \preceq H \prec F} f(H), \text{ by induction hypothesis} \\ &= (\bigodot_{a \preceq H \prec F} f(H)) \cdot f(F), \text{ since } f(F) = F^* \\ &= \bigodot_{a \preceq H \prec E} f(H) \end{aligned}$$

■

Let us describe the procedure *Continuations* to compute the list of the c-continuations $c_x(E)$ in a linear expression E according to Proposition 6.

Procedure Continuations(E: linear expression)
Step 1: Compute the set of links $f(F)$ in T_E.
Step 2: For each symbol x, concatenate the (non-unit) subexpressions $f(F)$ associated to subexpressions F which are on the path leading from the leaf labelled x to the root.

The complexity of the procedure Continuations lies on the following lemma.

Lemma 1 *Let E be a linear expression of size s and alphabetic width w. The size of a c-continuation in E is $O(s^2)$ and its alphabetic width is $O(ws)$. If E is starfree, the size of a c-continuation in E is $O(s)$ and its alphabetic width is $O(w)$.*

Proof. By Corollary 1, the c-continuation $c_a(E)$ of the symbol a in the linear expression E is a product of distinct subexpressions H_i of E, possibly reduced to a single subexpression or to 1. The size of a subexpression is bounded by s, and there are at most s subexpressions. Hence the size of a c-continuation in E is (roughly) bounded by s^2. The worst case is when E is a "nest of stars". For instance, let $E = a^{***}$. We have: $c_a(E) = a^* \cdot a^{**} \cdot a^{***}$. As illustrated by this example, the alphabetic width of the c-continuation can be equal to d times the alphabetic width of the expression, where d, the depth of the nest of stars is bounded by s. Hence the alphabetic width of a c-continuation in E is bounded by ws. On the opposite, if the expression is starfree, subexpressions H_i have disjoint positions, hence the size of a c-continuation is bounded by s and its alphabetic width is bounded by w. ■

Proposition 7 *Let E be a linear expression of size s and alphabetic width w. The procedure Continuations(E) computes the list of the c-continuations in E with an $O(ws^2)$ space and time complexity. If E is starfree, the complexity is $O(ws)$.*

Proof. Step 1 space and time complexity is $O(s)$. By Lemma 1, Step 2 space and time complexity is $O(ws^2)$ in the general case and $O(ws)$ when E is starfree. ■

4.2 C-continuations: identification

We now examine the way equivalent c-continuations can be identified.

Procedure	Identification(L: list of strings)
Step 1:	Sort L in lexicographic order.
Step 2:	For each string, compare it to the next one, and give a same class-name to identical strings.

Proposition 8 *Let L be a list of n strings with an $O(t)$ size over an alphabet of size k. The procedure Identification can be designed to yield the identification of the strings in L with an $O(nt+k)$ space and time complexity.*

Proof. Aho *et al.* algorithm [1] or Paige and Tarjan refinement [17] allow to sort a list of strings in lexicographic order with a time complexity $O(\sigma + k)$, where σ is the total sum of the sizes of the strings and k is the size of the alphabet. Hence an $O(nt + k)$ space and time complexity for Step 1. On the other hand, identical strings are given a same class-name by comparing each string to the following one, which leads to an $O(nt)$ time complexity. Finally, the space and time complexity of the whole procedure is $O(nt + k)$.

■

Corollary 2 *Let L be the list of the $w + 1$ c-continuations in a linear expression of size s. The procedure Identification yields the identification of strings in L with an $O(ws^2)$ space and time complexity.*

4.3 Final states: computation

By Definition 7 (resp. Definition 8), a state (x, c_x) of $\mathcal{C}_E$ (resp. a state $[c_x]$ of $\mathcal{C}_E/_\sim$) is final if and only if c_x is nullable. We compute $\lambda(c_x)$ by the formula: $\lambda(c_x) = \bigodot_{x \preceq F \prec E} \lambda(f(F))$.

Hence the procedure FinalStates (resp. FinalStatesQ) to compute the set of final states of $\mathcal{C}_E$ (resp. of $\mathcal{C}_E/_\sim$).

Procedure FinalStates(E: regular expression)
Step 1: Compute $\lambda(F)$, for each subexpression F of E.
Step 2: For each position x, deduce $\lambda(c_x)$ from values $\lambda(f(F))$ associated to subexpressions F on the path from the leaf labelled x to the root.

Procedure FinalStatesQ(E: regular expression)
Step 1: Compute $\lambda(F)$, for each subexpression F of E.
Step 2: For each $[c_x]$, deduce $\lambda(c_x)$ from values $\lambda(f(F))$ associated to subexpressions F on the path from the leaf labelled x to the root.

The complexity of these two procedures lies on the following lemma.

Proposition 9 *Let E be a regular expression of size s and alphabetic width w. The procedure FinalStates (resp. FinalStatesQ) computes the sets of final states of $\mathcal{C}_E$ (resp. of $\mathcal{C}_E/_\sim$) with an $O(w+s)$ space and an $O(ws)$ time complexity.*

Proof. Step 1 can be achieved by a recursive procedure with an $O(s)$ space and time complexity. Notice Step 2 needs no explicit computation of the c-continuations. Since a c-continuation is a product of $O(s)$ subexpressions of $\overline{E}$, $\lambda(c_x)$ can be computed with an $O(s)$ time complexity. Hence an $O(w)$ space complexity and an $O(ws)$ time complexity for Step 2. ■

4.4 Transitions: computation

Transitions deduce from the First sets of the subexpressions of E which appear in a c-continuation. Computation is carried through position sets associated to c-continuations and defined as follows:

Definition 9 *To each c-continuation c_x is associated the set $T_{c_x} = \{y \mid y \in Pos_E \text{ and } d_y(c_x) \neq 0\}$. To each class $[c_x]$ is associated the set $T_{[c_x]} = \{(a,y) \mid y \in Pos_E,\ d_y(c_x) \neq 0 \text{ and } a = h(y)\}$.*

4.4.1 Transitions in $\mathcal{C}_E$

By Definition 7, the transitions $((x,c_x),a,(y,c_y))$ of $\mathcal{C}_E$ are such that: $\delta((x,c_x),a) = \{(y,c_y) \mid h(y) = a \text{ and } d_y(c_x) \equiv c_y\}$, $\forall x \in Pos_E \cup \{0\}$ and $\forall a \in \Sigma$. Notice that $\mathcal{C}_E$ is homogeneous, i.e. $(y,c_y) \in \delta((x,c_x),a) \Rightarrow h(y) = a$, thus label a deduces from y.

By Theorem 3, we have: $d_y(c_x) \neq 0 \Rightarrow d_y(c_x) \equiv c_y$. Thus, instead of computing c-continuations $d_y(c_x)$, for all x, y in Pos_E, we compute sets T_{c_x}, for all x in Pos_E.

Proposition 10 *Let $c_x = H_1 \cdot H_2 \cdots H_l$ and let s, $1 \leq s \leq l$, the index such that $\lambda(H_i) = 1$ for all $1 \leq i < s$ and $\lambda(H_s) = 0$. We assume that $s = l$ if $\lambda(H_i) = 1$ for all $1 \leq i \leq l$. Then one has:*

$$T_{c_x} = \bigcup_{1 \leq i \leq s} First(H_i)$$

Proof. $T_{c_x} = \{y \mid y \in Pos_E \text{ and } d_y(c_x) \neq 0\}$. As $c_x = H_1 \cdot H_2 \cdots H_l$, by Definition 6 and Proposition 4, it comes:

$$\begin{aligned}
T_{c_x} &= \bigcup_{1\leq i\leq s} \{y \mid d_y(H_i) \neq 0 \text{ and } d_y(H_j) = 0 \text{ for } 1 \leq j < i\} \\
&= \bigcup_{1\leq i\leq s} \left(\{y \mid d_y(H_i) \neq 0\} \cap \{y \mid d_y(H_j) = 0 \text{ for } 1 \leq j < i\} \right) \\
&= \bigcup_{1\leq i\leq s} \left(First(H_i) \cap \{y \mid y \notin First(H_i), \text{ for } 1 \leq j < i\} \right) \\
&= \bigcup_{1\leq i\leq s} \left(First(H_i) \cap \left(Pos_E \setminus \bigcup_{1\leq j<i} First(H_j) \right) \right) \\
&= \bigcup_{1\leq i\leq s} \left(First(H_i) \setminus \bigcup_{1\leq j<i} First(H_j) \right) \\
&= \bigcup_{1\leq i\leq s} First(H_i)
\end{aligned}$$

■

Hence the procedure Transitions to compute the transitions of $\mathcal{C}_E$.

Procedure Transitions(E: regular expression)
Step 1: Compute the sets $First(H)$, for each subexpression H of E.
Step 2: For each position x, deduce the set T_{c_x} from the $First$ sets of the subexpressions $f(F)$ associated to subexpressions F on the path from the leaf labelled x to the root.

Proposition 11 *Let E be a regular expression of size s and alphabetic width w. The procedure Transitions computes the set of transitions of $\mathcal{C}_E$ with an $O(ws)$ space and an $O(w^2 s)$ time complexity.*

Proof. Step 1 can be achieved by a recursive procedure with an $O(ws)$ space and time complexity. Since a c-continuation is a product of $O(s)$ subexpressions of $\overline{E}$, and since the First set of a subexpression has an $O(w)$ size, T_{c_x} can be computed with an $O(ws)$ time complexity. Hence an $O(w^2)$ space complexity and an $O(w^2 s)$ time complexity for Step 2 and for the whole procedure. ■

4.4.2 Transitions in $\mathcal{C}_E/_\sim$

By Definition 8, transitions $([c_x], a, [c_y])$ of $\mathcal{C}_E/_\sim$ are such that: $\delta([c_x], a) = \{[c_y] \mid h(y) = a \text{ and } d_y(c_x) \equiv c_y\}$, $\forall [c_x] \in Q_\sim$ and $\forall a \in \Sigma$.

Notice that the automaton $\mathcal{C}_E/_\sim$ is not homogeneous: $c_z \in [c_y]$ does not imply $h(z) = h(y)$. This is the reason why computation cannot be carried through T_{c_x} sets, as for the automaton $\mathcal{C}_E$. Let us recall that by Definition 9 we have: $T_{[c_x]} = \{(a, y) \mid y \in Pos_E,\ d_y(c_x) \neq 0 \text{ and } a = h(y)\}$, for all $[c_x]$ in $Q_\sim$.

Assume that s is defined as in Proposition 10. We have: $T_{[c_x]} = \bigcup_{1 \leq i \leq s} \{(a, y) \mid y \in First(H_i) \text{ and } a = h(y)\}$.

Hence the procedure TransitionsQ to compute the transitions of $\mathcal{C}_E/_\sim$.

Procedure	TransitionsQ(E): E is a regular expression
Step 1:	Compute the sets $First(H)$, for each subexpression H of E.
Step 2:	For each $[c_x]$, deduce the set $T_{[c_x]}$ from the $First$ sets of subexpressions $f(F)$ associated to subexpressions F on the path from the leaf x to the root.

Proposition 12 *Let E be a regular expression of size s and alphabetic width w. The procedure TransitionsQ computes the set of transitions of $\mathcal{C}_E/_\sim$ with an $O(ws)$ space and an $O(w^2 s)$ time complexity.*

This proposition is a corollary of Proposition 11.

4.5 The algorithms *AlgoC* and *AlgoCQuotient*

We summarize previous results and give the complexity of the algorithms *AlgoC* and *AlgoCQuotient*.

Algorithm	*AlgoC*(E: regular expression)
Part 1:	Continuations(E)
Part 2:	FinalStates(E)
Part 3:	Transitions(E)

Notice the initial state of $\mathcal{C}_E$ is $(0, \overline{E})$.

Time complexity is $O(ws)$ for Part 2 (Proposition 9), $O(ws^2)$ for Part 1 (Proposition 7) and for Part 3 (Proposition 11). Space complexity is $O(w+s)$

for Part 2 (Proposition 9), $O(ws)$ for Part 3 (Proposition 11) and $O(ws^2)$ for Part 1 (Proposition 7). Hence an $O(ws^2)$ space and time complexity for the algorithm *AlgoC*.

Algorithm	*AlgoCQuotient*(E: regular expression)	
Part 1:	Continuations(E)	
	Identification(L), L is the list of c-continuations	
Part 2:	FinalStatesQ(E)	
Part 3:	TransitionsQ(E)	

Notice the initial state of $\mathcal{C}_E/_\sim$ is $[E]$.

By Corollary 2, and Propositions 9 and 12, space and time complexity of each Part of *AlgoCQuotient* is the same as in *AlgoC*. Hence the following theorem:

Theorem 7 *The Algorithm AlgoC (resp. AlgoCQuotient) computes the $\mathcal{C}_E$ automaton (resp. the $\mathcal{C}_E/_\sim$ automaton) of a regular expression E with an $O(\|E\| \cdot |E|^2)$ space and time complexity. If E is starfree the complexity is $O(\|E\| \cdot |E|)$.*

Notice that Algorithm *AlgoC* provides an $O(ws^2)$ construction of the position automaton, and Algorithm *AlgoCQuotient* an $O(ws^2)$ construction of the equation automaton of an expression.

4.6 Example

Let $E = ((x^*y)^*+x(x^*y)^*y)^*$. The quotient of the c-continuation automaton of E (see Figure 4) is computed by the program *ProgCQuotient*.

```
E=((x*.y)*+x.(x*.y)*.y)*
size=16 alphabetic-width=6 #stars=5
alphabet=xy01+.*

----------Computation of the c-continuations (alphabet "xy01+.*")
Before identification: 7 c-continuation(s)
l0=((x1*.y2)*+x3.(x4*.y5)*.y6)*
l1=x1*.y2.(x1*.y2)*.((x1*.y2)*+x3.(x4*.y5)*.y6)*
l2=(x1*.y2)*.((x1*.y2)*+x3.(x4*.y5)*.y6)*
l3=(x4*.y5)*.y6.((x1*.y2)*+x3.(x4*.y5)*.y6)*
l4=x4*.y5.(x4*.y5)*.y6.((x1*.y2)*+x3.(x4*.y5)*.y6)*
l5=(x4*.y5)*.y6.((x1*.y2)*+x3.(x4*.y5)*.y6)*
l6=((x1*.y2)*+x3.(x4*.y5)*.y6)*

After identification: 5 image(s) by mapping h
L0={1}: x*.y.(x*.y)*.((x*.y)*+x.(x*.y)*.y)*
```

```
L1={4}: x*.y.(x*.y)*.y.((x*.y)*+x.(x*.y)*.y)*
L2={0,6}: ((x*.y)*+x.(x*.y)*.y)*
L3={2}: (x*.y)*.((x*.y)*+x.(x*.y)*.y)*
L4={3,5}: (x*.y)*.y.((x*.y)*+x.(x*.y)*.y)*

----------The quotient of the c-continuation automaton
Initial state: 2
Final state(s): 2 3
Transitions:
State 0:   x: 0     y: 3
State 1:   x: 1     y: 4
State 2:   x: 0 4   y: 3
State 3:   x: 0 4   y: 3
State 4:   x: 1     y: 2 4
```

5 Quadratic algorithms: pseudo-continuations

We now present quadratic refinements. The worst-case space and time complexity of Algorithms *AlgoC* and *AlgoCQuotient* can be reduced to $O(s^2)$ according to three main improvements:

1. The star subexpressions of $\overline{E}$ are preprocessed: they are identified by names and their occurrences inside c-continuations are replaced by these names. We prove that the resulting strings can replace c-continuations in the identification step. Since they have an $O(s)$ size, we get an $O(s^2)$ computation of the set of states.

2. The First sets of subexpressions of E are computed via linkings over the syntax tree of E, with an overall $O(s)$ time complexity. This technique has already been used in the ZPC-algorithm [20, 18] to build the position automaton of an expression.

3. For each c-continuation c_x (resp. each class $[c_x]$), the set T_{c_x} (resp. $T_{[c_x]}$) is computed in $O(s)$ time as a disjoint union of First sets.

These improvements lead to the $O(s^2)$ Algorithms *AlgoCtoP* and *AlgoCtoE* to compute respectively the position automaton and the equation automaton.

5.1 An implicit computation of the c-continuations

According to Proposition 7, the computation of the c-continuations reduces to $O(ws)$ when E is starfree. On the other hand, the real objective of this computation rather is the identification of the c-continuations. Thus

a starfree expression E' is substituted to E, and pseudo-continuations in E' are substituted to c-continuations in E, so that the pseudo-continuations identification be equivalent to the c-continuations one. Pseudo-continuations will be introduced in the next subsection. We now explain how to compute E'.

5.2 Star expressions: preprocessing

Procedure	Stars(E: regular expression)
Step 1:	Process a left to right topdown traversal of $T(E)$, and store the star expressions in a list L.
Step 2:	Run the procedure Identification(L) to give a same star-name to identical strings.
Step 3:	Label star-links (s. t. $f(F) = F^*$) by the star-name of F^*.

Notice that the starfree expression E' is not explicitly computed: the aim is to substitute their star-names to star expressions involved in a c-continuation, and this can be achieved using the labels of star-links.

The complexity of this preprocessing is as follows:

Proposition 13 *Let E be a linear expression of size s and alphabetic width w. The procedure Stars(E) preprocesses the star expressions of $T(E)$ with an $O(s^2)$ space and time complexity.*

Proof. The size of a star expression is bounded by s, and there are at most $O(s)$ star expressions. Thus Step 1 space and time complexity is $O(s^2)$. On the other hand, by Proposition 2, Step 2 has an $O(nt + k)$ space and time complexity, where n is the number of strings, $O(t)$ their size, and k the size of the alphabet. Here the alphabet is $X = \Sigma_E \cup \{0, 1, +, \cdot, *\}$. Hence a time complexity $O(s^2 + |\Sigma_E|)$ i.e. $O(s^2)$ to achieve the Step 2. Links $f(F)$ are labelled via a traversal of $T(E)$ with an $O(s)$ time complexity. Thus Step 3 time complexity is $O(s^2)$. Finally the procedure Stars(E) has an $O(s^2)$ space and time complexity. ■

5.3 Pseudo-continuations

We now consider a regular expression E and its linearized version $\overline{E}$. Let us explain the way star-names are used inside c-continuations strings. We call

pseudo-continuation w.r.t. a position x in $\overline{E}$ the string $l_x(\overline{E})$ which deduces from the c-continuation $c_x(\overline{E})$ by substituting each star expression by its star-name.

More formally, let us denote $S(H)$ the star-name of the star expression H, and $s(H)$ the string associated to the expression H, and give the following definition:

Definition 10 *Let E be a regular expression and $c_x(\overline{E})$ the c-continuation w.r.t. x in $\overline{E}$. The pseudo-continuation w.r.t. x in $\overline{E}$ is the string $l_x(\overline{E})$ such that:*

$$\begin{aligned} c_x(\overline{E}) &= H_1 \cdot H_2 \cdot \ldots H_l \\ &\quad \textit{where } H_i \textit{ is a subexpression of } \overline{E},\ 1 \leq i \leq l \\ l_x(\overline{E}) &= A_1 \cdot A_2 \cdot \ldots A_l \\ \textit{where } A_i &= \begin{cases} S(H_i) & \textit{if } H_i \textit{ is a star expression} \\ s(H_i) & \textit{otherwise} \end{cases} \end{aligned}$$

We first show that c-continuations can be substituted by pseudo-continuations inside the identification process.

Let S_* be the alphabet of the star-names. A pseudo-continuation $l_x(\overline{E})$ is a string over the alphabet $Y = Pos_E \cup S_* \cup \{0, 1, +, \cdot\}$. Let us extend the mapping h from $Pos_E \cup S_*$ to Σ by $h(s) = s$ for all s in S_*.

Proposition 14 *Let x and y be positions in $Pos_E \cup \{0\}$. Then we have:*

$$h(l_x(\overline{E})) \equiv h(l_y(\overline{E})) \iff h(c_x(\overline{E})) \equiv h(c_y(\overline{E}))$$

Proof. ($\Rightarrow$) Obvious: identical star-names inside strings $h(l_x(\overline{E}))$ and $h(l_y(\overline{E}))$ necessary have the same expansion in $h(c_x(\overline{E}))$ and $h(c_y(\overline{E}))$.

($\Leftarrow$) This is due to the fact it is syntactically impossible for subexpressions F, G and H to verify $H^* \equiv F^* \cdot G^*$. ■

5.3.1 Pseudo-continuations: computation

Let us describe the procedure *Pseudocontinuations* to compute the list of the pseudo-continuations $l_x(\overline{E})$ in a regular expression E.

Procedure Pseudocontinuations(E: regular expression)

Step 1: Compute the set of links $f(F)$ in T_E.

Step 2: Run the procedure *Stars(E)* which preprocesses the star expressions and compute star-links labels.

Step 3: For each position x of $\overline{E}$, concatenate the strings A_i associated to subexpressions $f(F_i)$ such that F_i is on the path from the leaf labelled x to the root.

The complexity of the procedure Pseudocontinuations(E) deduces from Lemma 2.

Lemma 2 *Let E be a regular expression of size s and alphabetic width w. Then the alphabet of the pseudo-continuations in $\overline{E}$ has an $O(w+s)$ size. Pseudo-continuations have an $O(s)$ size.*

Proof. Pseudo-continuation are strings over the alphabet $Y = Pos_E \cup S_* \cup \{0, 1, +, \cdot\}$, where S_* is the alphabet of the star-names. Since the number of star expressions in E is bounded by s, the alphabet Y has an $O(w+s)$ size. On the other hand, in the product $l_x(\overline{E}) = A_1 \cdot A_2 \cdot \ldots A_l$, star-names have an overall size bounded by the number of star expressions in E, and the other strings have a total size bounded by s, since expressions they represent have disjoint alphabets. Thus a pseudo-continuation has an $O(s)$ size.

■

Proposition 15 *Let E be a regular expression of size s and alphabetic width w. The procedure Pseudocontinuations(E) computes the list of the pseudo-continuations in $\overline{E}$ with an $O(s^2)$ space and time complexity.*

Proof. Step 1 space and time complexity is $O(s)$. By Lemma 13, Step 2 space and time complexity is $O(s^2)$. By Lemma 2, Step 3 space and time complexity is $O(ws)$. Hence an $O(s^2)$ space and time complexity for the whole procedure. ■

5.3.2 Pseudo-continuations: identification

Proposition 16 *Let E be a regular expression of size s and alphabetic width w. The procedure Identification computes the identification of pseudo-continuations which graphically coincide on Σ with an $O(s^2)$ space and time complexity.*

Proof. There are $w+1$ pseudo-continuations in $\overline{E}$. By Lemma 2, they have an $O(s)$ size, and their alphabet has an $O(w+s)$ size. By Proposition 2, the procedure Identification runs on the list of pseudo-continuations with a space and time complexity $O(ws+w+s)$, thus $O(ws)$. ■

5.4 Transitions: computation

Transitions computation is improved by two features:

1. The First sets of subexpressions of E are computed via linkings over the syntax tree of E, with an overall $O(s)$ time complexity. This technique has already been used in the ZPC-algorithm [20, 18] to build the position automaton of an expression. We refer to these papers for further details.

2. For each c-continuation c_x (resp. each class $[c_x]$), the set T_{c_x} (resp. $T_{[c_x]}$) is computed in $O(s)$ time as a disjoint union of First sets.

The efficient computation of T_{c_x} and $T_{[c_x]}$ sets is based on the two following propositions:

Proposition 17 *Let $c_x = H_1 \cdot H_2 \cdot \ldots H_l$. Let r be an index such that $1 \leq r \leq l$, and H_r be a star-link such that $H_r = K^* = f(K)$. Then there exists a word u such that: $c_x = d_u(K) \cdot K^* \cdot H_{r+1} \ldots H_l$.*

The proof deduces from the property of c-derivatives we mentioned to explain Theorem 5: the structure of c-derivatives is such that it is equivalent to compute the c-derivatives of $F = d_u(\overline{E})$ and the c-derivatives of $G = \overline{h(F)}$.

As in Proposition10, we consider the index s, $1 \leq s \leq l$, such that $\lambda(H_i) = 1$ for all $1 \leq i < s$ and $\lambda(H_s) = 0$ (we assume that $s = l$ if $\lambda(H_i) = 1$ for all $1 \leq i \leq l$).

Proposition 18 *Let r, $1 \leq r \leq s$, be the greatest index such that H_r is a star-link. We assume that $r = 1$ if there is no star-link . Then, we have:*

$$T_{c_x} = \biguplus_{r \leq i \leq s} First(H_i)$$

Proof. By Proposition 10, for all state (x, c_x) of $\mathcal{C}_E$, $T_{c_x} = \bigcup_{1 \leq i \leq s} First(H_i)$. By Proposition 17, since H_r is a star-link, there exists a word u such that: $c_x = d_u(K) \cdot K^* \cdot H_{r+1} \ldots H_l$. Since $First(d_u(K)) \subseteq First(K^*)$, we get $T_{c_x} = \bigcup_{r \leq i \leq s} First(H_i)$. Moreover, since H_r is the "highest" star-link in T_{c_x}, this last union is a disjoint one. ■

We straightforwardly deduce:

$$T_{[c_x]} = \biguplus_{r \leq i \leq s} \{(a, y) \mid y \in First(H_i) \text{ and } a = h(y)\}$$

Hence the procedure Transitions2 (resp. Transitions2Q) to compute the transitions of $\mathcal{C}_E$ (resp $\mathcal{C}_E/_\sim$).

Procedure Transitions2(E: regular expression)
Step 1: Compute the sets $First(H)$, for H a subexpression of E, via ZPC-linkings over the syntax tree.
Step 2: For each c_x, compute T_{c_x} as a disjoint union of the $First$ sets of subexpressions $f(F)$ associated to subexpressions F on the path from the leaf x to the root.

The procedure Transitions2Q to compute the transitions of $\mathcal{C}_E/_\sim$ is similar.

Procedure Transitions2Q(E: regular expression)
Step 1: Compute the sets $First(H)$, for H a subexpression of E, via ZPC-linkings over the syntax tree.
Step 2: For each $[c_x]$, compute $T_{[c_x]}$ as a disjoint union of the First sets of the subexpressions $f(F)$ associated to subexpressions F on the path from the leaf x to the root.

Proposition 19 *Let E be a regular expression of size s and alphabetic width w. The procedure Transitions2 (resp. Transitions2Q) computes the set of transitions of $\mathcal{C}_E$ (resp. $\mathcal{C}_E/_\sim$) with an $O(w^2)$ space and an $O(ws)$ time complexity.*

Proof. Step 1 can be achieved with an $O(s)$ space and time complexity [20, 18]. For each c_x (resp. each $[c_x]$), computing indexes s and r is in $O(s)$ time. Each set T_{c_x} (resp. $T_{[c_x]}$) has an $O(w)$ size and deduces from a a disjoint union of First sets with an $O(w)$ space and time complexity. Hence an $O(w^2)$ space complexity and an $O(ws)$ time complexity for Step 2 and for the whole procedure. ■

5.5 The algorithms *AlgoCtoP* and *AlgoCtoE*

We summarize previous results and give the complexity of the algorithms *AlgoCtoP* and *AlgoCtoE*.

Algorithm	*AlgoCtoP*:
Part 1:	Pseudocontinuations(E)
Part 2:	FinalStates(E)
Part 3:	Transitions2(E)

Notice that the initial state is $(0, \overline{E})$.

Time complexity is $O(ws)$ for Part 2 (Proposition 9) and Part 3 (Proposition 19), and $O(s^2)$ for Part 1 (Proposition 15),

Space complexity is $O(w+s)$ for Part 2 (Proposition 9), $O(ws)$ for Part 3 (Proposition 19) and $O(s^2)$ for Part 1 (Proposition 15). Hence an $O(s^2)$ space and time complexity for the algorithm *AlgoC*.

Algorithm	*AlgoCtoE*(E: regular expression)
Part 1:	Pseudocontinuations(E); Identification(L),
	L is the list of pseudo-continuations
Part 2:	FinalStatesQ(E)
Part 3:	Transitions2Q(E)

Notice that the initial state is $[E]$.

By Corollary 16, and Proposition 9 and 19, space and time complexity of each Part of *AlgoCtoE* is the same as in *AlgoCtoP*.

Hence the following theorem:

Theorem 8 *The Algorithm AlgoCtoP (resp. AlgoCtoE) computes the $\mathcal{P}_E$ automaton (resp. the $\mathcal{E}_E$ automaton) of a regular expression E with an $O(|E|^2)$ space and time complexity.*

5.6 Example

Let $E = ((x^*y)^* + x(x^*y)^*y)^*$. The equation automaton of E (see Figure 2) is computed by the program *ProgCtoE*.

```
E=((x*.y)*+x.(x*.y)*.y)*
size=16 alphabetic-width=6 #stars=5
alphabet=xy01+.*
```

```
----------Star subexpressions preprocessing (alphabet "xy01+.*")
Before identification: 5 star(s)
s1=x1*
s2=(x1*.y2)*
s3=x4*
s4=(x4*.y5)*
s5=((x1*.y2)*+x3.(x4*.y5)*.y6)*

After identification: 3 star(s)
S1={5}: ((x*.y)*+x.(x*.y)*.y)*
S2={2,4}: (x*.y)*
S3={1,3}: x*

----------Computation of the pseudo-continuations (alphabet "xy01+.*SSS")
Before identification: 7 pseudo-continuation(s)
l0=S1
l1=S3.y2.S2.S1
l2=S2.S1
l3=S2.y6.S1
l4=S3.y5.S2.y6.S1
l5=S2.y6.S1
l6=S1

After identification: 5 image(s) by mapping h
L0={0,6}: S1
L1={3,5}: S2.y.S1
L2={2}: S2.S1
L3={4}: S3.y.S2.y.S1
L4={1}: S3.y.S2.S1

----------The equation automaton
Initial state: 0
Final state(s): 0 2
Transitions:
State 0:   x: 1 4    y: 2
State 1:   x: 3      y: 0 1
State 2:   x: 1 4    y: 2
State 3:   x: 3      y: 1
State 4:   x: 4      y: 2
```

6 Conclusion

The main result of this work is the following: the notion of c-continuation of a regular expression leads to an $O(|E|^2)$ space and time algorithm to compute the equation automaton of an expression, improving by an $O(\|E\|^3)$ factor the partial derivatives construction.
This new finite automaton construction is particularly interesting since it is as fast as the most efficient constructions of the position automaton, whereas the automaton it yields can be much smaller.

References

[1] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *Data Structures and Algorithms*. Addison-Wesley, Reading, MA, 1983.

[2] J.-M. Autebert et J.-M. Rifflet. Dérivations Formelles des Expressions Rationnelles : Un Programme de Calcul Automatique. *Rapport LITP* 78(14), 1978.

[3] V. Antimirov. Partial derivatives of regular expressions and finite automaton constructions. *Theoret. Comput. Sci.*, 155:291–319, 1996.

[4] D. Beauquier, J. Berstel, and P. Chrétienne. *Éléments d'Algorithmique*. Masson, Paris, 1992.

[5] A. Brüggemann-Klein, Regular Expressions into Finite Automata. *Theoret. Comput. Sci.*, 120(1993), 197–213.

[6] G. Berry and R. Sethi. From regular expressions to deterministic automata. *Theoret. Comput. Sci.*, 48(1):117–126, 1986.

[7] J. A. Brzozowski. Derivatives of regular expressions. *J. Assoc. Comput. Mach.*, 11(4):481–494, 1964.

[8] C.-H. Chang and R. Paige. From Regular Expressions to DFAs using Compressed NFAs, in Apostolico. Crochemore. Galil. and Manber. editors. *Lecture Notes in Computer Science*, 644(1992), 88-108.

[9] J. -M. Champarnaud and D. Ziadi. Canonical Derivatives, Partial Derivatives and Finite Automaton Constructions. *submitted*, 1999.

[10] J. -M. Champarnaud and D. Ziadi. From Mirkin's Prebases to Antimirov's Partial Derivatives. *submitted*, 1999.

[11] J. -M. Champarnaud and D. Ziadi. New Finite Automaton Constructions Based on Canonical Derivatives. In: S. Yu, ed., Proc. *CIAA'2000*, *Lecture Notes in Computer Science*, to appear.

[12] V. M. Glushkov. The abstract theory of automata. *Russian Mathematical Surveys*, 16:1–53, 1961.

[13] J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley, Reading, MA, 1979.

[14] S. Kleene. Representation of events in nerve nets and finite automata. *Automata Studies*, Ann. Math. Studies 34:3–41, 1956. Princeton U. Press.

[15] R. F. McNaughton and H. Yamada. Regular expressions and state graphs for automata. *IEEE Transactions on Electronic Computers*, 9:39–57, March 1960.

[16] B. G. Mirkin. An algorithm for constructing a base in a language of regular expressions. *Engineering Cybernetics*, 5:110–116, 1966.

[17] R. Paige and R. E. Tarjan. Three partition refinement algorithms. *SIAM J. Comput.*, 16(6), 1987.

[18] J.-L. Ponty, D. Ziadi and J.-M. Champarnaud, A new Quadratic Algorithm to convert a Regular Expression into an Automaton, In: D. Raymond and D. Wood, eds., Proc. *WIA'96*, *Lecture Notes in Computer Science*, Vol. 1260(1997) 109–119.

[19] S. Yu. Regular languages. In G. Rozenberg and A. Salomaa, editors, *Handbook of Formal Languages*, volume I, Word, Language, Grammar, pages 41–110. Springer-Verlag, Berlin, 1997.

[20] D. Ziadi, J.-L. Ponty, and J.-M. Champarnaud. Passage d'une expression rationnelle à un automate fini non-déterministe. *Bull. Belg. Math. Soc.*, 4:177–203, 1997.