Efficient Testing and Matching of Deterministic Regular Expressions

Efficient Testing and Matching of Deterministic Regular Expressions

B. Groz[a], S. Maneth[b,*]

[a]*LRI, Université Paris-Sud, France*
[b]*School of Informatics, University of Edinburgh, United Kingdom*

Abstract

A linear time algorithm is presented for testing determinism of a regular expression. It is shown that an input word of length n can be matched against a deterministic regular expression of length m in time $O(m + n \log\log m)$. If the deterministic regular expression has bounded depth of alternating union and concatenation operators, then matching can be performed in time $O(m + n)$. These results extend to regular expressions containing numerical occurrence indicators.

Keywords: deterministic regular expression, matching time complexity, testing determinism, numercial occurrence indicators

1. Introduction

Regular expressions are a fundamental concept in computer science. They were introduced in the 1950's by the mathematician Stephen Kleene [1]. Regular expressions allow to represent regular languages in a succinct and natural way. They have numerous applications, for instance, to search within a text editor (Ken Thomson's "ed" from 1971 already implements regular expression search, cf. [2]) or to find lines of a file that match a given expression (as in Unix's "grep" tool). Virtually all modern programming languages support regular expressions (directly or through libraries). Recent applications of regular expressions include validation of XML documents against XML Schemas, or network intrusion detection where network packages are compared against large collections of regular expressions to detect cyber attacks (see, e.g., [3]).

Modern applications, such as the two just mentioned, have a particular demand for *efficient matching* of an input word against a regular expression. For this reason, new efficient libraries have been developed (e.g. RE2 by Russ Cox [4]). In general however, matching requires non-linear running time. This is often unacceptable, especially when both the input word and the regular expression are large. For this

*Corresponding author
Email addresses: benoit.groz@lri.fr (B. Groz), smaneth@inf.ed.ac.uk (S. Maneth)

reason, the document processing community when in the 1960s developing the document markup language SGML [5] restricted the usage of regular expressions to so called *deterministic regular expressions*. Recent web standards such as XML [6] and XML Schema[7] have taken over this very same restriction (called "unique particle attribution" in XML Schema).

What is a deterministic regular expression? A regular expression is a well-bracketed word over terminal letters, union, and star. The union operator ("+") represents choice and the star operator ("$*$") represents (zero or more) repetitions. Brackets are used for grouping. As an example, consider the regular expression

$$e = a(a + b)^* b$$

This expression matches any word over the letters a and b that starts with an a and ends with a b. To understand how a word can be matched against an expression, consider the *positions* of the expression, i.e., all the occurrences of letters. The expression e contains four positions. Matching can be achieved by moving from positions to positions, while checking the letters of the current positions against the current letter in the input word. But, how exactly can we determine the admissible moves between positions? Consider the expression e. The only admissible start move is to the first a-position. From this position, we can move to the next a-position and then stay there (thus matching arbitrarily many a's). From any a-position we can also move to a b-position. From the first b-position we can stay, move to the second a-position, or move to the second b-position. The only admissible end position is the second b-position. Using these moves, let us match the word $w = ab$ against e: we start at the first a-position, obtaining a successful match against the first letter of w. From this first a-position, we can move to all positions of e except the first a. The two b-positions match successfully against the next letter of w. Call these positions the currently "active" positions. Since the second b-position is an admissible end position, w is successfully matched by e. It should be intuitively clear that this matching process is simplified if at any given moment, there is only at most one active position. The determinism restriction requires exactly this: from any position, the possible successor positions must all carry distinct labels. Thus, e is not deterministic because the first a-position admits two successor b-positions. Note that the language of e can, however, be captured by this deterministic regular expression:

$$d = aa^* b(a^* b)^*$$

In general, not every regular language can be captured by a deterministic regular expression. Anne Brüggemann-Klein and Derick Wood [8] show that, for instance, $(a+b)^* a(a+b)$ has no equivalent deterministic expression. (Note that Brüggemann-Klein and Wood use the term "1-unambiguous" to refer to deterministic regular expressions.) Thus, we lose expressive power when moving from regular to deterministic regular expressions.

How can it be decided for a given regular expression whether or not it is deterministic? There is a classical construction due to V. M. Glushkov [9], which

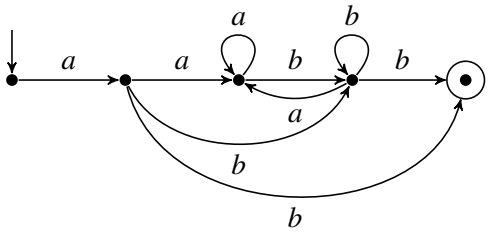

Figure 1: The Glushkov automaton A_e.

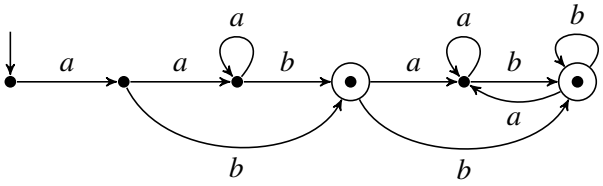

Figure 2: The Glushkov automaton A_d.

converts a regular expression e into a finite-state automaton. It turns out that the resulting automaton A_E (often called "Glushkov automaton" or "position automaton") is deterministic, if and only if the expression e is deterministic. The automaton A_E has one state for each position of e, plus an additional start state. If we can move from position p to p' and p' is labeled a, then A_E has an a-transition from state p to state p'. For every possible start move there is a corresponding transition from the start state of A_E. The admissible end positions are the final states of A_E. Figures 1 and 2 show the Glushkov automata for the regular expressions e and D of above. Clearly, A_E is not deterministic (viz. the second and fourth state from the left: they both have two outgoing b-transitions), while the automata A_D is deterministic.

What is the time complexity of *testing determinism* of a regular expression e? The best known algorithm constructs the Glushkov automaton and tests it for determinism. As shown by Brüggemann-Klein [10], this can be achieved in time $O(m^2)$, where m is the length of e. More precisely, the time complexity is $O(|\Sigma|m)$ where Σ is the set of letters occurring in e.

What is the complexity of *matching* a word (of length n) against a deterministic regular expression? We construct the Glushkov automaton and run it on the input word in time $O(m^2 + n)$. The most straightforward implementations of matching based on the traditional constructions of DFAs or NFAs have a running time of either $O(2^m + n)$ or $O(mn)$, respectively, for matching unrestricted regular expressions (there are of course many variants of automata-based matching algorithms which can achieve trade-offs between these bounds).

Naturally, the questions arise whether one can do (i) better than time $O(m^2)$ for

testing determinism, and (*ii*) better than time $O(m^2 + n)$ for matching. In this paper we present the first linear time algorithm for testing determinism. We also present linear time matching algorithms for several important subclasses of deterministic regular expressions. Our results concerning regular expressions are summarized as follows:

(1) Determinism of a regular expression (of length m) can be tested in time $O(m)$.

(2) A deterministic regular expression e can be matched in time $O(m+n)$ against an input word of length n, if

   (a) each letter occurs only a constant number of times in e, or

   (b) the maximal depth of alternating union and concatenation operations in e is constant.

(3) Star-free deterministic regular expressions can be matched against k input words (of lengths $n_1, \ldots, n_k$), in time $O(m + n_1 + \cdots + n_k)$.

(4) Arbitrary deterministic regular expressions can be matched in time $O(m + n \log\log m)$.

Before we explain the ideas behind these results, we give some examples of expressions which exhibit the quadratic running time of the previously best known method of testing determinism (namely to construct the Glushkov automaton). Consider the expression

$$f_m = (a_1?)(a_2?)\ldots(a_m?)$$

where $a_1, \ldots, a_m$ are distinct letters. The expression $(a?)$ matches a or the empty word. Thus, f_m matches words in which each a_i occurs at most once, and the index i increases from left to right. The Glushkov automaton for f_m (with $m = 4$) is

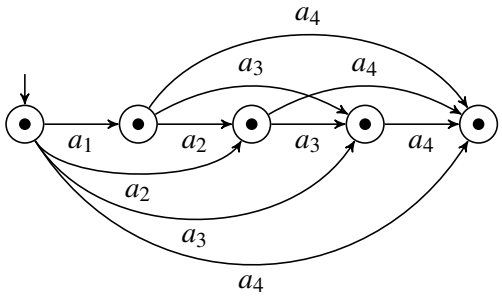

Figure 3: The Glushkov automaton for the expression f_4.

shown in Figure 3. Its number of transitions is $m(m + 1)/2$. As another example consider the expression

$$g_m = (a_1 + \cdots + a_m)(a_1 + \cdots + a_m)$$

which matches any two-letter word over the letters $a_1, \ldots, a_m$. The size (i.e., the number of transitions) of the Glushkov automaton for g_m is $m(m+1)$.

Testing Determinism. Since it is cost-prohibitive to construct the Glushkov automaton, the obvious choice is to construct the parse tree t_e of e and to devise an algorithm that tests determinism using t_e. It was observed already by Chang and Paige [11] and by Ponty, Ziadi, and Champarnaud [12] that it can be tested in a syntax-directed way over t_e, whether a position q of e follows a position p (i.e., whether or not there is a transition from q to p in A_e). Thus, we decorate t_e (during preprocessing) with additional information that allows to answer efficiently basic queries. For instance, checking if q follows p essentially amounts to checking if the lowest common ancestor (LCA) of q and p is a concatenation operator or if it has a star operator as its ancestor. Thus, we equip the parse trees with facilities allowing to efficiently answer LCA queries (see, e.g., the works by Harel and Tarjan and by Bender, Farach-Colton, Pemmasani, Skiena, and Sumazin [13, 14]) and pointers leading to the nearest star-ancestor. This is an example of the methodology that eventually allows us to test determinism in linear time. We use a variation of skeleta trees [15] to overcome yet another obstacle. Bojańczyk and Parys [15] use skeleta trees to give a linear time (data complexity) algorithm for evaluating regular XPath queries. Interestingly, one can construct one fixed XPath query that, given an expression e, tests determinism of e over t_e (we show this in Section 3.3).

Matching. Using a variation of skeleta trees that store additional information in their nodes, it is possible to match a word against a deterministic regular expression e by repeatedly asking so called "lowest colored ancestor" queries. A tree t can be preprocessed with a randomized algorithm in expected linear time so that arbitrary such queries can be answered in time $O(\log\log|t|)$ by the results of Muthukrishnan and Müller and by Farach and Muthukrishnan [16, 17]. This gives us time $O(m + n\log\log m)$ for matching a word of length m against e of length n. It is straightforward, using our data structures, to match in time $O(m+n)$ deterministic expressions in which each letter occurs only a constant number of time. It is more involved to obtain an $O(m+n)$ time algorithm for deterministic expressions with constant alternation depth of concatenation and union operators. For this, the parse tree t_e is decomposed into paths. Each such path is rooted by a concatenation or by a union operator, and ends in a leaf (position). Roughly speaking, the matching algorithm simulates the transitions of the Glushkov automaton, by repeatedly jumping between the different paths. Through an amortized running time analysis we obtain the linear bound. Finally, to match several words against a star-free deterministic regular expression, we devise an algorithm that builds skeleton trees dynamically, while traversing the expression once. We expect that our linear time algorithm can well be applied in practise.

Note that the alternation depth is small in practice: Grijzenhout's large collection of (thousands of) real-world DTDs [18] does not contain a single regular expression with alternation depth larger than 4. Also in other domains, such as network intrusion detection, we expect expressions to have small alternation depth.

Expressions with Numerical Occurrence Indicators. Regular expressions that appear in practical applications often use a richer syntax than the one we have discussed until now. In particular, it is common to use a shorthand to denote the k-fold repetition of a subexpression e. In XML Schema [7] this shorthand is called "numeric occurrence indicator", and it is of the form $e^{i..j}$, where $i < j$ are non-negative integers. The expression $e^{i..j}$ is a shorthand for $(e^i + e^{i+1} + \cdots + e^j)$. Such expression appear, for instance, also in the "interval expressions" of POSIX and in the "patterns" of the programming language Perl [19, 20]. A recent study [21] shows they are widely used in practice: a majority of expressions feature such indicators in *RegExLib*, the main collection of regular expressions on the web, and in *Snort*, a system for web intrusion detection. We extend our results of before to regular expressions with numeric occurrence indicators. For such expressions, two forms of determinism have been studied: weak and strong (see Kilpeläinen and Tuhkanen [22]). Essentially, weak determinism only checks that we can determine the position in the regular expression while matching a word against a regular expression, whereas strong determinism additionally checks that there is no ambiguity about how many iterations of every subexpression were required: expression $(a^{1..3})^{2..2}$ is only weakly deterministic (a^5 can be decomposed into $(a^3)(a^2)$ or $(a^2)(a^3)$) whereas $(a^{2..2})^{3..4}$ is strongly deterministic.

Our results concerning regular expressions with numerical occurrence indicators are summarized as follows. Let e be a regular expression (of size m) with numeric occurrence indicators.

(5) Weak and strong determinism of e can be tested in time $O(m)$.

(6) If e is strongly deterministic, then matching e against a word of length n can be performed in time $O(m + n)$ if (a) each letter occurs only a constant number of times in e, or if (b) the maximal depth of alternating union and concatenation operations in e is constant.

(7) If e is strongly deterministic, then e can be matched in time $O(m+n \log\log m)$.

A preliminary version of this paper was presented at PODS 2012 [23].

2. Regular Expressions

Let Σ be an alphabet, i.e., a finite non-empty set of symbols. By Σ^* we denote the set of all words over Σ. The empty word is denoted by ε. *Regular expressions over* Σ are defined by the following grammar:

$$e :- a(e) \odot (e)(e) + (e)(e)?(e)^*,$$

where $a \in \Sigma$, $\odot$ represents concatenation, $+$ union, $?$ choice, and $*$ the Kleene star. The language $L(e)$ of e is defined recursively as usual (see, e.g., [24]): $L(a) = a$, $L((e_1) \odot (e_2)) = \{uw \mid u \in L(e_1), w \in L(e_2)\}$, $L((e_1) + (e_2)) = L(e_1) \cup L(e_2)$, $L((e)?) = L(e) \cup \{\varepsilon\}$, and $L((e)^*) = \{w^k \mid w \in L(e), k \geq 0\}$, where $w^0 = \varepsilon$ and

$w^{n+1} = ww^n$. We observe that the languages $\{\varepsilon\}$ and $\emptyset$ cannot be expressed with this syntax (but these trivial languages are irrelevant for our study). We say that e is *nullable* if $\varepsilon \in L(e)$. In expressions, we often omit $\odot$ symbols, and omit parentheses as well when there is no ambiguity (e.g., around words in Σ). We require that our alphabet Σ contains the symbols # and \$, and we require of our regular expressions e that:

(R1) $e = (\#e')\$$ and # and \$ do not appear in e',

(R2) $((e')^*)^*$ does not appear in e,

(R3) if $(e')?$ appears in e, then $\varepsilon \notin L(e')$.

An arbitrary regular expression can be changed easily (in linear time) into an equivalent one of the required form. In fact, our condition is a weaker version of the star normal form by Brüggemann-Klein [25]. Note that # and \$ are tacitly present and required, but, for better readability, are omitted in most examples.

We identify a regular expression with its parse tree (as illustrated in Figure 4), and define the *positions* $Pos(e)$ of e as the leaves of e whereas N_e denotes the set of all nodes from e. A position of the regular expression is thus viewed as a "location" in the tree. For a node $n \in N_e$ we denote by e/n the subexpression of e rooted at n. Every tree t is implemented as a pointer structure, where $Lchild_t(n)$ (resp. $Rchild_t(n)$) returns the left (resp. right) child of node n in t and $parent_t(n)$ returns the parent of n in t. The pointers return *Null* if the respective node does not exist. In particular, for unary nodes $Rchild_t(n)$ returns *Null*. We denote by $lab_t(n)$ the label of n in t, and by $\preccurlyeq_t$ the (reflexive) ancestor relationship in t. If $m \preccurlyeq_t n$ then we also say that n is a descendant of m. Thus, each node is ancestor and descendant of itself. We finally define for each node n the pointer $pStar(n)$ which points to the lowest $*$-labeled ancestor of n. Our algorithms will exploit the property that all $pStar(n)$ pointers can be computed in linear time through a depth-first traversal of the tree.

The *size* of a tree t, denoted $|t|$, is the number of nodes in t, whereas the *depth* of t, denoted $depth(t)$, is the length of the path (number of edges) from the root to the deepest node in t. Our restrictions (R2) and (R3) guarantee that $|e|$ is linear in $|Pos(e)|$. We denote by $\overline{e}$ the regular expression obtained from e by marking the i-th position (when traversing the regular expression (tree) from left to right) with subscript i. For instance, $\overline{(a \odot b)^*(a + b)} = (a_1 \odot b_2)^*(a_3 + b_4)$. We denote by $\overline{\Sigma}$ the set of symbols obtained from Σ by adding subscripts below symbols. By definition, the trees $\overline{e}$ and e are the same except for this relabeling of positions so we will identify each position from $\overline{e}$ with its corresponding position in e.

Given a position p of e, $Follow_e(p)$ is the set of positions that may follow p in e:

$$Follow_e(p) = \{q \mid \exists u, v \in \overline{\Sigma}^*.u \odot lab_{\overline{e}}(p) \odot lab_{\overline{e}}(q) \odot v \in L(\overline{e})\}. \tag{1}$$

The expression e is *deterministic* if for all $p, q, q' \in Pos(e)$ with $q, q' \in Follow_e(p)$: $q \neq q'$ implies that $lab_e(q) \neq lab_e(q')$. Whenever the regular expression or the tree is clear from context, we drop the subscript and write *Follow*, *lab*, and $\preccurlyeq$.

Example 2.1. Let $e_1 = (ab + b(b?)a)^*$ and $e_2 = (a^*ba + bb)^*$. Denote by $p_1, \ldots, p_5$ the positions of e_1 in left-to-right order, and by $q_1, \ldots, q_5$ those of e_2. Then $\overline{e_1} = (a_1b_2 + b_3(b_4?)a_5)^*$ and $Follow_{e_1}(p_3) = \{p_4, p_5\}$. Similarly, $\overline{e_2} = (a_1^*b_2a_3 + b_4b_5)^*$, and $Follow_{e_2}(q_3) = \{q_1, q_2, q_4\}$. Expression e_1 is deterministic, while e_2 is non-deterministic because $lab_{e_2}(q_2) = lab_{e_2}(q_4) = b$.

2.1. Structure of Regular Expressions

The *First* and *Last*-positions of a regular expression e are

$$\begin{aligned} First(e) &= \{p \mid \exists u \in \overline{\Sigma}^*.lab_{\overline{e}}(p) \odot u \in L(\overline{e})\} \\ Last(e) &= \{p \mid \exists u \in \overline{\Sigma}^*.u \odot lab_{\overline{e}}(p) \in L(\overline{e})\}. \end{aligned}$$

We also define, for a node n of e, $First(n)$ and $Last(n)$ as $First(e/n)$ and $Last(e/n)$, respectively. Note that $First(n)$ and $Last(n)$ are non-empty for every node n of e. For instance, for the expression e_0 in Figure 4, $First(n_2) = \{p_1, p_2\}$ and $Last(n_2) = \{p_5\}$.

Given two nodes u, v of e, let $LCA(u, v)$ denote the lowest common ancestor of u and v in e. The next lemma was stated before, e.g., in [11, 12], but not in terms of *LCA*.

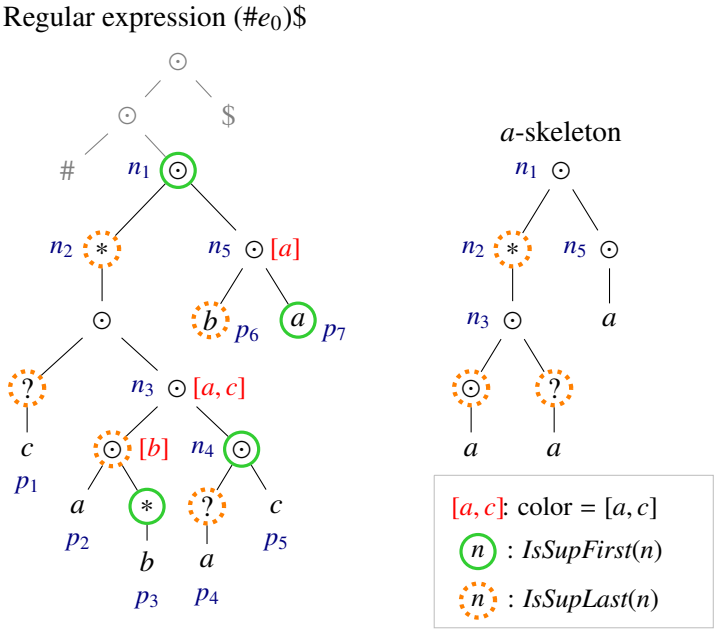

Figure 4: Expression $e_0 = (c?((ab^*)(a?c)))^*(ba)$.

Lemma 2.2. Let $p, q \in Pos(e)$ and $n = LCA(p, q)$. Then $q \in Follow(p)$ iff

(1) $lab(n) = \odot$, $q \in First(Rchild(n))$, $p \in Last(Lchild(n))$, or

(2) $q \in \mathit{First}(s)$ and $p \in \mathit{Last}(s)$ where s is the lowest $*$-labeled ancestor of n.

Lemma 2.2 says that there are only two ways in which positions follow each other: (1) through a concatenation, or (2) through a star. We write $q \in \mathit{Follow}^{\odot}_{e}(p)$ if (1) is satisfied, and $q \in \mathit{Follow}^{*}_{e}(p)$ if (2) is satisfied. For instance, in e_0 (Figure 4), we have $p_4 \in \mathit{Follow}^{\odot}_{e_0}(p_3)$ and $p_1 \in \mathit{Follow}^{*}_{e_0}(p_5)$. Note, however, that there may exist positions p and q that satisfy simultaneously (1) and (2).

It was also observed earlier, e.g., [11, 12, 26], that First and Last-sets (and nullability) can be defined in a syntax-directed way over the parse tree of e. For instance, if $\mathit{lab}(n) = \odot$ and $\mathit{Lchild}(n)$, $\mathit{Rchild}(n)$ are non-nullable then $\mathit{First}(n) = \mathit{First}(\mathit{Lchild}(n))$ and $\mathit{Last}(n) = \mathit{Last}(\mathit{Rchild}(n))$. We define now the Boolean properties *IsSupFirst* and *IsSupLast* for every node n, where n' denotes $\mathit{parent}(n)$:

$\mathit{IsSupFirst}(n)$ iff $\mathit{lab}(n') = \odot, n = \mathit{Rchild}(n')$, and $\mathit{Lchild}(n')$ is non-nullable

$\mathit{IsSupLast}(n)$ iff $\mathit{lab}(n') = \odot, n = \mathit{Lchild}(n')$, and $\mathit{Rchild}(n')$ is non-nullable.

If $\mathit{IsSupFirst}(n)$ then the First-set changes at n's parent: $\mathit{First}(\mathit{parent}(n)) \cap \mathit{First}(n) = \varnothing$. Otherwise the First-set of n's parent is a superset: $\mathit{First}(\mathit{parent}(n)) \supseteq \mathit{First}(n)$. For instance, in e_0 (Figure 4), *IsSupFirst* holds at n_4 since $\mathit{First}(n_3) = \{p_2\}$ and $\mathit{First}(n_4) = \{p_4, p_5\}$. This explains the name "*IsSupFirst*": a node with this property is "maximal" with respect the First-sets of its direct descendants (without the property). The same holds for *IsSupLast* and *Last*. We define for any node n, the pointers $\mathit{SupFirst}(n)$ and $\mathit{SupLast}(n)$ as the lowest ancestors x of n such that $\mathit{IsSupFirst}(x)$ and $\mathit{IsSupLast}(x)$, respectively. Recall that by (R1), $e = (\#e')\$$; this implies that for every node of e', both $\mathit{SupFirst}(n)$ and $\mathit{SupLast}(n)$ are defined. We do not define pointers *SupFirst* and *SupLast* for the "help nodes" of e that are not in e' (such as the root node of e). Note however, that *IsSupFirst* holds at the node n_1 in Figure 4 because of the phantom position #. From now on we omit in examples the two phantom positions and their parent nodes. Together with ancestor queries, the pointers *SupFirst* and *SupLast* allow to check membership in First and Last, according to the following Lemma:

Lemma 2.3. Let $p \in \mathit{Pos}(e)$ and $n \in N_e$.

(1) $p \in \mathit{First}(n)$ iff $\mathit{SupFirst}(p) \preccurlyeq n \preccurlyeq p$, and

(2) $p \in \mathit{Last}(n)$ iff $\mathit{SupLast}(p) \preccurlyeq n \preccurlyeq p$.

It is well-known, see [13, 14], that arbitrary LCA queries on a tree t can be answered in constant time, after preprocessing of t in linear time. And ancestor queries a fortiori: $n \preccurlyeq n'$ can be checked, e.g., by testing if $\mathit{LCA}(n, n') = n$. For positions p and q, define the Boolean $\mathit{checkIfFollow}(p, q)$ as true iff $q \in \mathit{Follow}(p)$.

Theorem 2.4. After preprocessing of e in $O(|e|)$ time, $\mathit{checkIfFollow}(p, q)$ can be answered in constant time for every $p, q \in \mathit{Pos}(e)$.

Proof. We preprocess e for LCA queries, and compute the three pointers $SupLast(n)$, $SupFirst(n)$, and $pStar(n)$ for each node n of e . After this linear time preprocessing we can compute $checkIfFollow(p, q)$ in constant time for any p and q: first obtain $n = LCA(p, q)$. By Lemma 2.2 we should return *true* if and only if one of the following two conditions are satisfied: (1) $lab(n) = \odot$, $q \in First(Rchild(n))$, $p \in Last(Lchild(n))$, or (2) $q \in First(s)$ and $p \in Last(s)$ where s is the lowest $*$-labeled ancestor of n. By Lemma 2.3, (1) is equivalent to $lab(n) = \odot$, $SupFirst(q) \preccurlyeq Rchild(n)$, $Rchild(n) \preccurlyeq q$, $SupLast(p) \preccurlyeq Lchild(n)$ and $Lchild(n) \preccurlyeq p$. Furthermore, (2) is equivalent to $SupFirst(q) \preccurlyeq n'$ and $SupLast(p) \preccurlyeq n'$ where $n' = pStar(n)$. These conditions can all be checked in constant time. □

The following technical lemmas state relationships between positions and their *SupFirst* and *SupLast* nodes.

Lemma 2.5. Let $p, q \in Pos(e)$ such that $q \in Follow_e(p)$. Let $n = LCA(p, q)$.

(1) $parent(SupFirst(q)) \preccurlyeq n$ and

(2) $parent(SupLast(p)) \preccurlyeq n$.

Proof. Let p, q, n satisfy the Lemma's conditions. If Case (1) of Lemma 2.2 holds, then $q \in First(Rchild(n))$ so $parent(SupFirst(q)) \preccurlyeq n$. If not, then Case (2) of Lemma 2.2 holds, hence $parent(SupFirst(q)) \preccurlyeq pStar(n) \preccurlyeq n$. In both cases we proved that $parent(SupFirst(q)) \preccurlyeq n$. Point (2) can be proved similarly. □

Lemma 2.6. Let $p, q \in Pos(e)$ such that $q \in Follow_e(p)$. If $SupLast(p) \preccurlyeq parent(SupFirst(q))$ then $SupFirst(q)$ is nullable.

Proof. Before we proceed with the proof, we first observe that for any pair of nodes $y \preccurlyeq x$ in e such that x is nullable and every node on the path from x to y (y excluded) satisfies neither *IsSupFirst* nor *IsSupLast*, then all nodes on this path from x to y are nullable. This property follows immediately from the definitions, by induction. Let $p, q \in Pos(e)$ such that $q \in Follow(p)$ and $SupLast(p) \preccurlyeq parent(SupFirst(q))$, and let $n = LCA(p, q)$. By Lemma 2.5, $SupLast(p)SupLast(p) \preccurlyeq parent(SupFirst(q)) \preccurlyeq n$. Assume first that $q \in Follow^{\odot}(p)$. Then $lab(n) = \odot$, $Lchild(n) \preccurlyeq p$ and $Rchild(n) \preccurlyeq q$. By definition of *SupLast*, the nodes on the path from p to $SupLast(p)$ (except $SupLast(q)$, of course) do not satisfy *IsSupLast*. This is true in particular for $Lchild(n)$, hence $Rchild(n)$ is nullable. If $SupFirst(q)$ is $Rchild(n)$, we just proved it is nullable. Otherwise, $SupFirst(q)$ is an ancestor of n by Lemma 2.5. In that case, there are no nodes satisfying *IsSupFirst* on the path from q to $SupFirst(q)$ by definition of *SupFirst*, so that in particular $Lchild(n)$ is nullable. Consequently, n is nullable, and there are no nodes satisfying *IsSupFirst* or *IsSupLast* between n and $SupFirst(q)$. Therefore, $SupFirst(q)$ is nullable.

The case $q \in Follow^{*}(p)$ is handled similarly: $pStar(n)$ is nullable and satisfies $SupFirst(q) \preccurlyeq pStar(n) \preccurlyeq n$. Moreover there are no nodes satisfying *IsSupFirst* or *IsSupLast* between n and $SupFirst(q)$, except $SupFirst(q)$. Thus, $SupFirst(q)$ is nullable. □

2.2. Regular Expressions with Numeric Occurrence Indicators

Definition and Notations. Regular expressions occurring in XML Schema may contain numeric occurrence indicators. Following the definitions of Kilpeläinen and Tuhkanen in [22], regular expressions with counting extend regular expressions with $e^{i..j}$ where $i \leq j$ are non-negative integers and j may also equal ∞. The expression $e^{m..n}$ denotes the union of all $L(e \odot e \cdots \odot e)$, where e appears k times in the latter expression and $m \leq k \leq n$. The syntax of regular expressions with numeric occurrence indicators is therefore:

$$e :- a(e) \odot (e)(e) + (e)(e)?(e)^{m..n}$$

with $a \in \Sigma$, $m \in \mathbb{N}$, $n \in \mathbb{N}_{>0} \cup \{\infty\}$, $m \leq n$ and $n \geq 2$. We also use e^m as a shorthand for $e^{m..m}$.

We first observe that the Kleene star is unnecessary when numeric occurrence indicators are allowed: e^* can be expressed as $e^{0..\infty}$. Moreover, we can also assume that in every numeric occurrence indicator $m..n$ the value of n is at least 2: this is because $e^{0..1}$ is equivalent to $e?$, and $e^{1..1}$ to e. We again assume the presence of the extra nodes # and \$ at the beginning and end of the expression. We extend the definition of the First- and Last-sets, and of nullability in a natural fashion. In the sequel, we identify each node of the parse tree with the subexpression it represents. An *iterative expression* (or node) is an expression of the form $s = (e)^{m..n}$. Finally, we denote the bounds of s by $min(s) = m$ and $max(s) = n$.

Matching with Numeric Occurrences. In the absence of numeric indicators, our matching algorithms map each letter of an input word w to some position of an expression e. Then $w \in L(e)$ if and only if there is a mapping for which each pair of consecutive positions belongs to the *Follow* relation, and such that the positions corresponding to the first and last symbol belong to $First(e)$ and $Last(e)$ respectively. To determine whether $w \in L(e)$ in presence of numeric occurrences, however, it is not enough to map input symbols to positions and check independently pairs of consecutive positions. Indeed, if we match a word against $a^{2..2}b$, the b-labeled position can follow the a-labeled position only after two a-labeled positions have been read. In order to take this phenomenon into account, our matching algorithms in presence of numeric occurrences map each input symbol to a configuration. A configuration consists of a position together with a valuation function c mapping each iterative subexpression to some integer. For technical reasons we initialize this counter to the value 1; and the value of the counter remains 1 for every iterative subexpression that is not an ancestor of the current position. For every iterative subexpression s that is an ancestor of the current position, however, the counter has value k if we have just matched $k-1$ consecutive occurrences of s, and we are currently matching the kth while processing the position.

We first extend the *Follow* relation to configurations. We will then show in Lemma 2.7 that transitions between configurations (i.e., which relation can follow another one) play a similar role as the transitions of the Glushkov automaton

between positions (i.e., the Follow relation) in the setting without numeric occurrences. These transitions between configurations are further illustrated in Figure 5 of Example 2.8.

We first define $Follow_e^{\odot}$ and $Follow_e^{s}$ that extend the *Follow* relation. Let p and q be positions, $n = LCA(p, q)$, let $s \preccurlyeq n$ be an iterative subexpression of e, and let c, c' be two valuations for the counters.

(1) $(q, c') \in Follow_e^{\odot}(p, c)$ iff $lab(n) = \odot$, $SupLast(p) \preccurlyeq n$, $SupFirst(q) \preccurlyeq n$, every iterative subexpression $n \preccurlyeq x \preccurlyeq p$ satisfies $min(x) \leq c(x)$, and c' is obtained from c by resetting (to "1") the counters of every iterative expression $n \preccurlyeq x \preccurlyeq p$.

(2) $(q, c') \in Follow_e^{s}(p, c)$ iff the following five conditions are satisfied: (1) $c(s) < max(s)$, (2) $SupLast(p) \preccurlyeq s$, (3) $SupFirst(q) \preccurlyeq s$, (4) every iterative expression $s \prec x$ satisfies $min(x) \leq c(x)$, and (5) c' is obtained from c by incrementing $c(s)$ and resetting (to "1") the counters of every iterative expression $s \prec x \preccurlyeq p$.

Those relations are illustrated in Example 2.8. We say that (q, c') *follows* (p, c) iff $(q, c') \in Follow_e^{\odot}(p, c)$ or $(q, c') \in Follow_e^{s}(p, c)$. We observe that p, q and $s \in \{\odot\} \cup N_e$ determine c': there is at most one c' such that $(q, c') \in Follow_e^{s}(p, c)$.

Lemma 2.7. A non-empty word $w = a_1 \dots a_n$ is accepted by a regular expression e with numeric occurrence indicators iff there exist n positions $p_1, \dots, p_n$ and valuations $c_1, \dots, c_n$ satisfying the following conditions:

(1) $p_1 \in First(e)$ and $c_1(x) = 1$ for every iterative subexpression x of e

(2) $p_n \in Last(e)$ and $c_n(x) \geq min(x)$ for every $x \preccurlyeq p_n$

(3) $lab(p_i) = a_i$ for each $i \leq n$

(4) (p_{i+1}, c_{i+1}) follows (p_i, c_i) for every $i < n$

Proof. The correctness follows from the results of Gelade, Gyssens, and Martens [27]: A configuration can follow another one if and only if there is a transition between the corresponding configurations of the counter automaton built from the expression in [27], whereas Conditions 1 and 2 guarantee that configurations (p_1, c_1) and (p_n, c_n) are respectively initial and final in the counter automaton. The correctness thus follows from [27, Theorem 9] where the equivalence between an expression e and its counter automaton is established through a relatively straightforward but lengthy induction on e. □

Example 2.8. Figure 5 shows one possible way of matching the word a^8b against the expression $e_5 = ((a^{2..3} + b)^2)^2 b$ (taken from [22]). Configurations are represented vertically, with the position on top and the value of the three counters below. For instance, if we define c_1 as the first valuation $(x \to 1, x' \to 1, x'' \to 1)$ and c_2 as the second $(x \to 2, x' \to 1, x'' \to 1)$ we have $(a_1, c_2) \in Follow_e^{x'}(a_1, c_1)$.

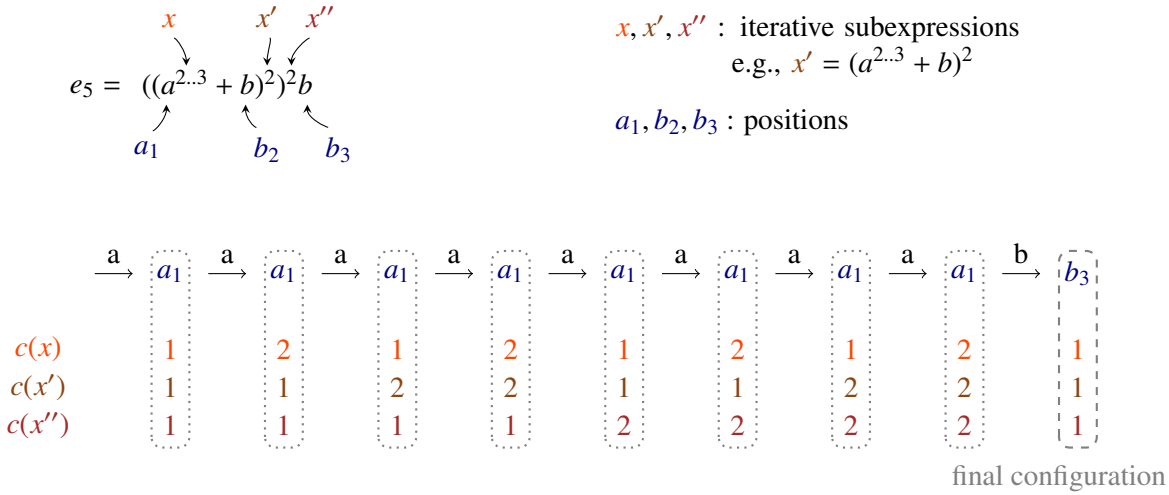

Figure 5: Sequence of consecutive configurations while matching e_5.

Determinism with Numeric Occurrence Indicators. In the absence of numeric occurrences, an expression is deterministic if and only if for each position p and letter a there is at most one a-labeled position q following p. In presence of numeric occurrences determinism comes in two flavours: *weak* and *strong*. Strong determinism requires determinism with respect to configurations whereas weak determinism only requires determinism with respect to positions. More formally, a regular expression e is *weakly deterministic* iff for each letter a and configuration (p, c), there exists at most one position p' such that there exists c' with (p', c') following (p, c). A regular expression e is *strongly deterministic* iff for each letter a and configuration (p, c), there exists at most one position p' and valuation c' such that (p', c') follows (p, c). In other words, strong determinism dictates that (in addition to weak determinism) for every value of p, q and c, there is at most one $s \in \{\odot\} \cup N_e$ (and therefore one c') such that $(q, c') \in \mathit{Follow}^s_e(p, c)$. We observe that for strongly deterministic expressions, $(q, c') \in \mathit{Follow}^s_e(p, c)$ implies $\mathit{max}(x) = c(x)$ for every $s \prec x \preccurlyeq n$.

Example 2.9. For both definitions of determinism, $e = (ab)^{2..2}a(b+d)$ is deterministic, but $e' = (ab)^{1..2}a$ is not, because $w = aba$ can lead to two a-labeled positions in e'. To see that nested iterative expressions can interact with each other, consider the expression $e_5 = ((a^{2..3} + b)^2)^2 b$ from Example 2.8. This expression is not even weakly deterministic because the word $w = a^8 b$ can lead to the two b-labeled positions: to the second one (b_3) if we decompose it into $((a^2)^2)^2 b$ as illustrated in Figure 5, and to the first one (b_2) with decomposition $(a^3)^2 a^2 b$.

Our adoption of a "constructive" definition of weak and strong determinism results in non-standard definitions, but one can check easily that these definitions are reformulations on the parse tree of the standard characterizations of determin-

ism with numeric occurrences [28, 27]. In particular the equivalence between the usual semantic definition and determinism for automata with counters is presented in [27]. It should be noted that, in the absence of numeric occurrences, every weakly deterministic expression can be converted into a strongly deterministic one in polynomial time, but weakly deterministic expressions are exponentially more succinct and more expressive in presence of numeric occurrences [27].

3. Testing Determinism

To test determinism we need to check for every $a \in \Sigma$ and positions $q \neq q'$ labeled a whether there exists a p such that q and q' follow p. The challenge of a linear time algorithm is to deal with the quadratically many candidate pairs (q, q').

3.1. Candidate Pair Reduction

We define the following condition:

(P1) for all $q \neq q'$ in $Pos(e)$, $SupFirst(q) = SupFirst(q')$ implies $lab(q) \neq lab(q')$.

Clearly, if (P1) is false then e is not deterministic. To see this, let $q \neq q'$ and $n = SupFirst(q) = SupFirst(q')$. Since the First- and Last-sets of any node are non-empty, there exists a p in $Last(Lchild(parent(n)))$. Note that $parent(n) = LCA(p, q) = LCA(p, q')$. By Lemma 2.2, $q, q' \in Follow_e(p)$, and hence by definition of determinism, $lab(q) \neq lab(q')$. Testing (P1) in linear time is straightforward: during one traversal of e we group the positions with the same *SupFirst*-pointer; for each group we check that all contained positions have distinct labels. This can easily be achieved in linear time, using an adapted bucket sorting algorithm. Therefore we assume from now on that (P1) is true.

We define the set $FollowAfter_e(n)$ which extends *Follow* to internal nodes n of e:

$$FollowAfter_e(n) = \{q \mid n \not\leftarrow q \text{ and } \exists p \in Last(n).q \in Follow_e(p)\}.$$

We next present a data structure used in our algorithm testing determinism. For each position p labeled a, we

- assign *color* a to the node $parent(SupFirst(p))$ and
- say that position p is a *witness* for color a in the node $parent(SupFirst(p))$.

Observe that each node may be assigned several colors, but, since (P1) holds, each node has at most one witness per color. In Figure 4, node n_3 has colors a and c. The witness for color a (resp. c) in n_3 is p_4 (resp. p_5).

We say that a node $n \in N_e$ has *class* a if n has color a, or n is a position labeled a, or n is the lowest common ancestor of two nodes of class a. The *a-skeleton* t_a of e consists of all nodes n of class a plus their *SupLast* and *pStar* nodes (as defined in Section 2). The node labels in t_a are taken over from e, and the tree structure is inherited from e: n' is the left (resp. right) child of n in t_a if (1) n' is in the subtree

of the left (resp. right) child of n in e, (2) $n \preccurlyeq n'$, and (3) there is no n'' in t_a with $n \preccurlyeq n'' \preccurlyeq n'$. If a node has no left (resp. right) child defined in this way, then the corresponding pointer is set to *Null*. Note that a node in t_a can be labeled $\odot$ or $+$ and have its left (or right) child point to *Null*. Figure 4 presents a regular expression and its a-skeleton.

Lemma 3.1. The collection of a-skeleta for all $a \in \Sigma$ can be computed in time $O(|e|)$.

Proof. The size of the a-skeleton is linear in the number of positions labeled a in e. Hence the size of the collection of a-skeleta is linear in $|e|$. The skeleta can be constructed in linear time by simply applying LCA repeatedly, inserting each position from e in left-to-right order using the linear preprocessing so that the LCA of two nodes of e is obtained in constant time. This construction is detailed in Proposition 4.4 of [15]. □

In the a-skeleton t_a, we equip each node n with three pointers: $\mathit{Witness}(n, a)$, $\mathit{FirstPos}(n, a)$, and $\mathit{Next}(n, a)$. For every node n in t_a,

- if n has color a then $\mathit{Witness}(n, a)$ is the witness for color a in n (and is undefined otherwise)
- $\mathit{FirstPos}(n, a)$ is the position p labeled a such that $p \in \mathit{First}(n)$ if it exists (and is undefined otherwise); note that property (P1) guarantees that there is at most one such position p
- $\mathit{Next}(n, a)$ is the set of all positions in $\mathit{FollowAfter}_e(n)$ labeled a.

Constructing the data structures *FirstPos* and *Witness* is straightforward: *Witness* is built simultaneously with the a-skeleton; *FirstPos* can for instance be computed in a single bottom-up traversal of each a-skeleton, using pointers *SupFirst* from e and ancestor queries in e. Let n be the root node of the a-skeleton. Then $\mathit{BuildNext}(a, n, \varnothing)$ in Algorithm 1 constructs the data structure $\mathit{Next}(n', a)$ for all nodes n' of the a-skeleton.

Lemma 3.2. Executing $\mathit{BuildNext}(n, a, \varnothing)$ for each $a \in \Sigma$ and root node n of t_a takes in total time $O(|e|)$. If any call returns *false* then e is non-deterministic. Otherwise, the set $\mathit{Next}(n, a)$ defined during the execution consists of all positions in $\mathit{FollowAfter}_e(n)$ labeled a, for $n \in N_{t_a}$ and $a \in \Sigma$.

Proof. The $O(|e|)$ time is achieved because (1) *BuildNext* is called at most m times, where m is the number of nodes of all skeleta, and $m \in O(|e|)$ by Lemma 3.1, and (2) each line of the algorithm runs in constant time because $|Y| \leq 2$ at each call, due to Line 10. To see the correctness consider the execution along a path in t_a. If at Line 7 the current node n has an ancestor u labeled $*$ with no *IsSupLast*-node on their path, then Y contains $\mathit{FirstPos}(u, a)$; if n is in the left subtree of

Algorithm 1: Computing $Next(n, a)$, if e is deterministic.

```
procedure BuildNext(a : Σ, n : Node, Y : Set(Node)) : Bool
1   if IsSupLast(n)
2       then Y ← ∅
3   if n is the left child in t_a of a ⊙-node and
4     n has a right sibling n' in t_a and
5     (¬IsSupLast(n) or parent_{t_a}(n) = parent_e(n))
6       then Y ← Y ∪ {FirstPos(n', a)}
7   Next(n, a) ← {p ∈ Y | n ⋠ p}
8   if lab(n) = ∗
9       then Y ← Y ∪ {FirstPos(n, a)}
10  if |Y| > 2
11      then return false
12  if Lchild_{t_a}(n) = Null
13      then return true
14      else B ← BuildNext(a, Lchild_{t_a}(n), Y)
15  if Rchild_{t_a}(n) = Null
16      then return B
17      else return B ∧ BuildNext(a, Rchild_{t_a}(n), Y)
end procedure
```

an ancestor u labeled $\odot$ with no *IsSupLast*-node on their path, and n has a right sibling n' in t_a, then Y contains $FirstPos(n', a)$. These conditions imply that the set defined in Line 7 holds all a-labeled positions in $FollowAfter_e(n)$. Clearly, e is non-deterministic if $|Y| > 2$ in Line 10. □

We define another condition:

(P2) for every $a \in \Sigma$ and $n \in N_{t_a}$, $Next(n, a)$ contains at most one element.

Clearly, (P2) can be tested in linear time (for instance by incorporating it into Algorithm 1). If (P2) is false, then e is non-deterministic. Thus, from now on we assume that both (P2) and (P1) are true. We identify $Next(n, a)$ with q if $Next(n, a) = \{q\}$ and let it be undefined otherwise.

Lemma 3.3. Let $p, q \in Pos(e)$ with $lab_e(q) = a$. If $q \in Follow_e(p)$ then the lowest ancestor n of p having color a exists and satisfies $q = Witness(n, a)$ or $q = FirstPos(n, a)$ or $q \in Next(n, a)$.

Proof. Lemma 2.5 states that a position q labeled a that follows p is a witness for color a in *some ancestor* of p. Thus, if two positions labeled a follow p, then each of them is witness for color a in ancestors of p. By Lemma 2.2, Lemma 2.5 (1), and Lemma 3.2: $q = Witness(n, a)$ if $Rchild(n) \preccurlyeq_e q$, $q = FirstPos(n, a)$ if $Lchild(n) \preccurlyeq_e q$, and $q = Next(n, a)$ if $n \not\preccurlyeq_e q$. □

From Lemma 3.3 and the definition of (P1) and (P2) we obtain the following result.

Lemma 3.4. The expression e is non-deterministic iff (P1) or (P2) is false, or there exist $a \in \Sigma$, $n \in N_{t_a}$ of color a, and $q, q' \in \{\mathit{FirstPos}(n,a), \mathit{Witness}(n,a), \mathit{Next}(n,a)\}$ such that $q \neq q'$ and $\mathit{Follow}_e^{-1}(q) \cap \mathit{Follow}_e^{-1}(q') \neq \varnothing$.

Proof. By definition, e is non-deterministic if the above conditions are satisfied. Reciprocally, if e is non-deterministic and does not satisfy properties (P1) and (P2), there exists $a \in \Sigma$ and positions p, q, q' such that $q \neq q'$, $\mathit{lab}_e(q) = \mathit{lab}_e(q') = a$ and $q, q' \in \mathit{Follow}_e(p)$. Let then n denote the lowest ancestor of p with color a. By Lemma 3.3, q and q' belong to $\{\mathit{FirstPos}(n,a), \mathit{Witness}(n,a), \mathit{Next}(n,a)\}$. □

3.2. Determinism Testing Algorithm

To check determinism using Lemma 3.4 we need to check for $a \in \Sigma$ and $n \in N_{t_a}$ of color a, and for every pair of distinct positions q and q' in $\{\mathit{FirstPos}(n,a), \mathit{Witness}(n,a), \mathit{Next}(n,a)\}$ whether or not

$$\mathit{Follow}_e^{-1}(q) \cap \mathit{Follow}_e^{-1}(q') \neq \varnothing.$$

Three combinations can occur for a position p:

(1) $\mathit{Witness}(n,a)$ and $\mathit{Next}(n,a)$ follow p, or

(2) $\mathit{Witness}(n,a)$ and $\mathit{FirstPos}(n,a)$ follow p, or

(3) $\mathit{FirstPos}(n,a)$ and $\mathit{Next}(n,a)$ follow p.

The third combination, however, reduces to the other two and therefore does not need to be considered:

Lemma 3.5. Let n, p, a be such that $N = \mathit{FirstPos}(n,a)$ and $F = \mathit{Next}(n,a)$ follow p as in Combination (3) above. Then there exists some node $n' \in N_{t_a}$ of color a such that one of the two following properties is satisfied:

(1) $F = \mathit{FirstPos}(n',a)$ and $N = \mathit{Witness}(n',a)$

(2) $F = \mathit{Witness}(n',a)$ and $N = \mathit{FirstPos}(n',a)$

Proof. Let N and F denote the nodes $\mathit{Next}(n,a)$ and $\mathit{FirstPos}(n,a)$, respectively, and let n_N and n_F denote the parent of their respective *IsSupFirst*-nodes. By definition, both n_N and n_F are strict ancestors of n. Furthermore, $n \preccurlyeq F$ but $n \not\preccurlyeq F$ by definition so that $N \neq F$, which by (P1) implies $n_N \neq n_F$. If $n_F \preccurlyeq n_N \preccurlyeq n$, $F = \mathit{FirstPos}(n_N,a)$ as $F \in \mathit{First}(x)$ for every $\mathit{SupFirst}(F) \preccurlyeq x \preccurlyeq n$. Furthermore, $N = \mathit{Witness}(n_N,a)$ by definition, so this configuration corresponds to combination (2) with $n' = n_N$. Otherwise $n_N \preccurlyeq n_F \preccurlyeq n$, in which case N is one of $\mathit{FirstPos}(n_F,a)$ or $\mathit{Next}(n_F,a)$. Furthermore, $F = \mathit{Witness}(n_F,a)$, which concludes our proof by setting $n' = n_F$. □

We next illustrate the two combinations:

Example 3.6 (Combination (1)). Let $e = (c(b?a?))a$, and let n be the parent of the c node in e. Thus, n is of color a, with the left a in e as witness. Clearly e is non-deterministic: take p as the c position, then both $Witness(n, a)$ and $Next(n, a)$ follow p. The same holds for the expressions $e' = (c(a?b?))a$ and $e'' = (c(b?a)^*)a$. However, expression $e''' = (c(b?a))a$ is deterministic; this is because n's right subtree is non-nullable, which prevents that $Next(n, a)$ and $Witness(n, a)$ both follow a same position p.

It is not hard to see, and is formally shown in the proof of Theorem 3.8, that Combination (1) occurs if and only if the right-child of n is nullable. Combination (2) can only occur if there is a $*$-node $S = pStar(n)$ above n, and $SupLast(n)$ is above this node S.

Example 3.7 (Combination (2)). Let $e = (a(b?a))^*$ and let n be the parent of the first a-position. As we can see, this expression is deterministic. This is for a similar reason as before: because the right child of n is non-nullable. If we consider $e' = (a(b?a?))^*$ then this expression is indeed non-deterministic and it holds that both $FirstPos(n, a)$ and $Witness(n, a)$ follow position p, where p is for instance the b-position. Thus, combination (2) requires that the right child of n is nullable, and also that $FirstPos(S, a) = FirstPos(n, a)$. The latter guarantees that on the path from S to $FirstPos(n, a)$ there is nothing non-nullable "to the left", and hence, that $FirstPos(n, a)$ follows the same position p that $Witness(n, a)$ follows.

To check determinism of e we check (P1), (P2), and then we execute for every $a \in \Sigma$ and every node n with color a, $CheckNode(n, a)$ of Algorithm 2; if any call returns *false*, then e is non-deterministic.

Theorem 3.8. Determinism of a regular expression e can be decided in time $O(|e|)$.

Proof. Let S, W, N, and F denote the sets of nodes $pStar(n)$, $Witness(n, a)$, $Next(n, a)$, and $FirstPos(n, a)$ respectively. Since (P1) and (P2) can be tested in $O(|e|)$ time, it suffices, by Lemma 3.4, to prove the following two statements.

(i) $Follow_e^{-1}(W) \cap Follow_e^{-1}(N) \neq \varnothing$ iff $Rchild_e(n)$ is nullable and $N \neq Null$,

(ii) $Follow_e^{-1}(W) \cap Follow_e^{-1}(F) \neq \varnothing$ iff $F \neq Null$, $S \neq Null$, $Rchild_e(n)$ is nullable, $FirstPos(S, a) = F$, and $SupLast(n) \preccurlyeq S$.

Let us prove statement (i) first. If $N \neq Null$ and $Rchild_e(n)$ is nullable then $Lchild_e(n)$ is not an *IsSupLast*-node. Therefore any position in $Last(Lchild_e(n))$ belongs to $Follow_e^{-1}(W) \cap Follow_e^{-1}(N)$. For the only-if direction, let q be a position in $Follow_e^{-1}(W) \cap Follow_e^{-1}(N)$. Then in particular $N \neq Null$. Node n is a strict ancestor of q since $q \in Follow_e^{-1}(W)$ and $n = parent_e(SupFirst(W))$. As q belongs to $Follow_e^{-1}(N)$, $SupLast(q)$ is an ancestor of n. This implies that $Rchild(n)$

is nullable according to Lemma 2.6, since $Rchild(n) = SupFirst(W)$ and W follows q.

Proof of (ii): If $F \neq Null$, $S \neq Null$, $Rchild_e(n)$ is nullable, $FirstPos(S, a) = F$, and $SupLast(n) \preccurlyeq S$, then any q in $Last(Lchild(n))$ is in $(Follow_e^{\odot})(W) \cap (Follow_e^{*})^{-1}(F)$. Conversely, let q be a position in $Follow_e^{-1}(W) \cap Follow_e^{-1}(F)$. As q belongs to $Follow_e^{-1}(W)$, node n is a strict ancestor of q. If $Rchild_e(n) \preccurlyeq_e q$ then $q \in (Follow_e^{*})^{-1}(F)$, hence $FirstPos(S, a) = F$ and $SupLast(n) \preccurlyeq S$, and furthermore $SupLast(q) \preccurlyeq S$, so that $Rchild_e(n)$ is nullable according to Lemma 2.6. Assume now that $Lchild_e(n)$ is an ancestor of q and let $x = LCA(q, F)$. As an ancestor of both q and F, $Lchild_e(n)$ is an ancestor of x. Furthermore, there is no *IsSupLast*-node between q and $Lchild_e(n)$, except possibly $Lchild_e(n)$, and there is no *IsSupFirst*-node between F and $Lchild_e(n)$. Consequently, x is non-nullable because $Lchild_e(n)$ is, and, there is no $*$-labeled node between x and $Lchild_e(n)$. Hence $q \notin (Follow_e^{\odot})^{-1}(F)$, and, more generally, $Follow_e^{-1}(W) \cap (Follow_e^{\odot})^{-1}(F)$ is empty. This means that $q \in (Follow_e^{*})^{-1}(F)$. Thus $S = pStar(x)$ is not $Null$, satisfies $FirstPos(S, a) = F$, and is an ancestor of n since there is no $*$-labeled node between x and $Lchild_e(n)$. Accordingly, $SupLast(q) \preccurlyeq S$ and hence $Rchild_e(n)$ is non-nullable. □

Algorithm 2: Checking determinism.

```
procedure CheckNode(n : Node, a : Σ) : Bool
1   F ← FirstPos(n, a)
2   S ← pStar(n)
3   if Rchild_e(n) is nullable and
4     (Next(n, a) ≠ Null or
5      (FirstPos(S, a) = F and SupLast(n) ≼ S))
6       then return false
7   return true
end procedure
```

3.3. Alternative Determinism Test

Determinism of e can be formulated as follows:

$$\neg(\exists p, p_1, p_2 \in Pos(e).\ lab_e(p_1) = lab_e(p_2) \wedge p_1 \in Follow_e(p) \wedge p_2 \in Follow_e(p)).$$

A natural question arises: Is there a logic that allows to capture determinism and, at the same time, has efficient model checking that yields a procedure for checking determinism in linear time? The answer is positive: It is possible with $\mathcal{X}_{reg}^{=}$, the language of Regular XPath expressions with data equality tests for binary trees with data values as defined in [15].

Trees with data values allow to store with every node its label, drawn from a finite set, and additionally, a data value, drawn from an infinite set. Regular

XPath allows to navigate the nodes of the tree using regular expressions of simple steps (e.g., parent to the left child) and filter expressions. Filter expressions with data equality allow essentially to test whether two nodes have the same data value. In [15] Bojańczyk and Parys show that an $\mathcal{X}^{=}_{reg}$-expression φ can be evaluated over a tree t in time $2^{O(|\varphi|)}|t|$.

We wish to construct an $\mathcal{X}^{=}_{reg}$-expression φ_{det} that captures determinism and whose size is constant i.e., does not depend on the regular expression e. The main challenge is to handle position labels of e that can be drawn from an alphabet of arbitrary size. This is accomplished by: 1) storing the labels of positions of e as data values and 2) using data equality to check whether two positions have the same label.

Theorem 3.9. There exists an $\mathcal{X}^{=}_{reg}$-expression φ_{det} such that for any regular expression e, φ_{det} is satisfied in e if and only if e is deterministic.

Proof. We present only the construction of the formula φ_{det}. Let *IsSupFirst* and *IsSupLast* denote $\mathcal{X}^{=}_{reg}$-expressions that are satisfied only in *IsSupFirst*- and *IsSupLast*-nodes, respectively.

$$
\begin{aligned}
D &= (\Downarrow/[\,\mathsf{not}\ \mathit{IsSupFirst}])^*/[\,\mathsf{not}\Downarrow] \\
U &= ([\,\mathsf{not}\ \mathit{IsSupLast}]/\Uparrow)^* \quad F = ([\mathrm{lab}()=\odot])/\text{to-right}/D \\
\varphi_{\odot\odot} &= \Downarrow^*/[\,\mathsf{not}\ \mathit{IsSupLast}]/\text{from-left}/[F=(U/\text{from-left}/F)] \\
\varphi_{**} &= \Downarrow^*/[\mathrm{lab}()=*]/ \\
&\qquad [D=(U/[\mathit{IsSupFirst}]/\Uparrow/U/[\mathrm{lab}()=*]/D)] \\
\varphi_{\odot *} &= \Downarrow^*/[\,\mathsf{not}\ \mathit{IsSupLast}]/\text{from-left}/ \\
&\qquad [(\text{to-right}/[\mathit{IsSupFirst}]/D)=(\Uparrow/U/[\mathrm{lab}()=*]/D)] \\
&\qquad \cup \Downarrow^*/[\mathrm{lab}()=*]/[D=(U/\text{from-left}/F)] \\
\varphi_{P_1} &= \Downarrow^*/[(\text{to-left}/[\,\mathsf{not}\ \mathit{IsSupFirst}]/D) = \\
&\qquad\qquad (\text{to-right}/[\,\mathsf{not}\ \mathit{IsSupFirst}]/D)] \\
\varphi_{det} &= [\,\mathsf{not}(\varphi_{P_1}\ \mathsf{or}\ \varphi_{\odot\odot}\ \mathsf{or}\ \varphi_{\odot *}\ \mathsf{or}\ \varphi_{*\odot}\ \mathsf{or}\ \varphi_{**})].
\end{aligned}
$$

When evaluated from some node n, the auxiliary expression D retrieves all the positions in $\mathit{First}(n)$. Similarly, F selects the ancestors x of n such that $\mathit{Last}(n) \subseteq \mathit{Last}(x)$. Using these expressions, φ_{P_1} checks if (P1) is violated in e and the expression $\varphi_{\ell\ell'}$ for $\{\ell, \ell'\} \subseteq \{*, \odot\}$ checks whether there exist two distinct positions p_1 and p_2 of e such that $\mathit{lab}(p_1) = \mathit{lab}(p_2)$ and $(\mathit{Follow}^{\ell}_{e})^{-1}(p_1) \cap (\mathit{Follow}^{\ell'}_{e})^{-1}(p_2) \neq \varnothing$. □

3.4. Testing Determinism in Presence of Numeric Occurrence Indicators

In order to deal with those interactions between iterations, Kilpeläinen and Tukhanen [22] define the *flexibility* of f in e, for every iterative subexpression f of e. They explain how to annotate, in time $O(|e|)$, every node n of e with a Boolean value indicating the flexibility of n. Essentially, flexible expressions are the only iterative expressions we have to consider when assessing determinism (in particular $*$ expressions are flexible).

3.4.1. Flexibility

We first recall from [22, 29] the definitions and useful properties of flexibility. An illustration of flexibility will be provided in Example 3.13 at the end of this subsection.

Definition 3.10 ([22, 29]). Let e be a non-nullable regular expression with numeric occurrence indicators. A subexpression f of e is *flexible in* e if and only if it is marked as flexible when executing $markFlexible(e, 1)$, where *markFlexible* is the procedure from [29] reproduced in Algorithm 3 below.

We adopt for Algorithm 3 the convention that $\infty \times r = \infty$ for any rational r, and of course $\infty \times \infty = \infty$.

Algorithm 3: Marking flexible subexpressions of e

procedure *markFlexible*(f : Subexpression of e, N : Integer) : Rational number or ∞

1 **case** $f = x$ for $x \in \Sigma$: **return** 1
2 **case** $f = g? : markFlexible(g, N)$ **return** ∞
3 **case** $f = g + h$: **return** $max(markFlexible(g, N), markFlexible(h, N))$
4 **case** $f = g \odot h$:
5 **if** $\epsilon \in L(h)$
6 **then** $f_g \leftarrow markFlexible(g, N)$
7 **else** $f_g \leftarrow markFlexible(g, 1)$
8 **if** $\epsilon \in L(g)$
9 **then** $f_h \leftarrow markFlexible(h, N)$
10 **else** $f_h \leftarrow markFlexible(h, 1)$
11 **if** $\epsilon \in L(g)$ **and** $\epsilon \in L(h)$ **return** ∞
12 **if** $\epsilon \in L(g)$ **and** $\epsilon \notin L(h)$ **return** f_h
13 **if** $\epsilon \notin L(g)$ **and** $\epsilon \in L(h)$ **return** f_g
14 **if** $\epsilon \notin L(g)$ **and** $\epsilon \notin L(h)$ **return** 1
15 **case** $f = g^{m..n}$:
16 $f_g \leftarrow markFlexible(g, N \times n)$
17 **if** $m < n$ **or** $f_g \geq (N \times n)/(N \times n - 1)$
18 **then** Mark f as flexible in e
19 **return** $f_g \times (n/m)$

Flexibility is illustrated in Example 3.13 below. Kilpeläinen and Tukhanen also define the relation $foll_e$ which essentially adapts the *Follow* relation of Equation 1 to expressions with numeric occurrence indicators, taking flexibility into account. The relation depends on the global expression e, but we drop the subscript to simplify the notations, since its value is always e in the following:

Definition 3.11 ([22]). Let e be a regular expression with numeric occurrence indicators. The relation $foll(f) \subseteq Pos(f) \times Pos(f)$ is defined for each subexpression f of e inductively as follows.

(1) If $f = a$ ($a \in \Sigma$), then $foll(f) = \emptyset$.

(2) If $f = g?$, then $foll(f) = foll(g)$.

(3) If $f = g + h$, then $foll(f) = foll(g) \cup foll(h)$.

(4) If $f = g \odot h$, then $foll(f) = foll(g) \cup foll(h) \cup (Last(g) \times First(h))$.

(5) If $f = g^{m..n}$ then

$$foll(f) = \begin{cases} foll(g) \cup (Last(g) \times First(h)) & \text{if } f \text{ is flexible in } e \\ foll(g) & \text{otherwise.} \end{cases}$$

We also define the subrelation $foll^{\odot}$ replacing Case 5 with $foll(f) = foll(g)$.

The determinism of regular expressions with numeric occurrence indicators can be characterized in terms of the following proposition.

Proposition 3.12 ([22]). Let e a regular expression with numeric occurrence indicators. Then e is non-deterministic if and only if there are two distinct positions $x, y \in Pos(e)$ such that $lab(x) = lab(y)$ and:

(1) $(z, x), (z, y) \in foll(e)$ for some position $z \in Pos(e)$, or

(2) $(z, x) \in foll(g)$, $y \in First(g)$ and $z \in Last(g)$ for position z and some subexpression of the form $f = g^{m..n}$ in e.

Example 3.13. Let $e = \left((b?)a^{2..3}\right)^{3..3} b$. We label positions in e as b_1, a_2, b_3 from left to right. In this expression, the subexpression $h = a^{2..3}$ is marked flexible. So is also the subexpression $f = \left((b?)a^{2..3}\right)^{3..3}$, because at line 17 of Algorithm 3 we have $N = 1, m = n = 3$ so that $3/2 \geq (N \times n)/(N \times n - 1) = 3/2$. As a consequence, (a_2, b_3) belongs to $foll(f)$ and hence to $foll(e)$. As (a_2, b_3) also belongs to $foll(e)$, e is non-deterministic by Proposition 3.12.

The intuition behind the notion of flexibility is the following: as f is flexible, we can "loose track" of how many iterations through f have been processed; the word a^6b, for instance, can be decomposed into $(a_2)^3(a_2)^3b_1$ or $(a_2)^2(a_2)^2(a_2)^2b_3$. In $e' = \left((b?)a^{2..3}\right)^{2..2} b$, a contrario, f is not flexible, so that e' is deterministic.

In Proposition 3.12 we essentially distinguish two situations that provide a witness for non-determinism. A third situation was actually considered in [22, 29]: when both x and y belong to $First(e)$ they also form a witness for non-determinism, but this situation is ruled out in our setting by the introduction of the virtual nodes # and $.

3.4.2. Results that Carry over from Standard Expressions

We do not modify the definitions of *SupFirst* and *SupLast* in presence of numeric occurrence indicators. Lemma 2.3 still holds and Lemma 2.5 can be adapted as follows: if q belongs to $foll(p)$ then we have the two following properties: (1) $parent(SupFirst(q)) \preccurlyeq p$ and (2) $parent(SupLast(p)) \preccurlyeq q$. Lemma 2.6, however, does not hold in presence of numeric occurrence indicators: consider for instance the positions p and q with label b and c in $(a((b+c)^{2..3}))d$. Then $SupFirst(q)$ is non-nullable, although $q \in foll(p)$ and $SupLast(p) \preccurlyeq parent(SupFirst(q))$. However, we can weaken Lemma 2.6:

Lemma 3.14. Let p and q be two positions of e such that $q \in Follow^{\odot}(p)$. If $SupLast(p) \preccurlyeq parent(SupFirst(q))$ then $SupFirst(q)$ is nullable.

Proof. Let $n = LCA(p, q)$. From $q \in Follow^{\odot}(p)$ we deduce that $SupLast(p) \preccurlyeq n$ and $SupFirst(q) \preccurlyeq n$, hence n is nullable. If $SupLast(p) \preccurlyeq parent(SupFirst(q))$ then there are no *SupFirst* nor *SupLast* nodes between n and $SupFirst(q)$, which implies that $SupFirst(q)$ also is nullable. □

We again observe that an expression must satisfy property (P1) to be deterministic, and henceforth assume the expression satisfies (P1) because the property can be tested in linear time. The definitions of $FirstPos(n, a)$ and $Witness(n, a)$ are not modified. In the definition of $Next(n, a)$ we need only a minor modification: star expressions are replaced by flexible iterative expressions: instead of testing $lab(n) = *$ at Line 8 of Algorithm 1, one tests if n is a flexible iterative expression in e. Then every deterministic regular expression again satisfies property (P2), which is tested within Algorithm 1. Instead of *pStar* we maintain a pointer $NextFlex(n)$ storing the closest ancestor of n that is a flexible iterative expression and such that $SupFirst(n) \preccurlyeq NextFlex(n)$ and $SupLast(n) \preccurlyeq NextFlex(n)$. If there is no such ancestor then $NextFlex(n) = Null$

Then Lemma 3.3 carries over (using *foll* instead of *Follow*). However, Proposition 3.12 tells us that one must also consider non-flexible iterations in addition to the *foll* relation (Case 2). We therefore define $NextNFlex(n, a)$ to take those into account. For every node n with color a, $NextNFlex(n, a)$ is defined as the lowest ancestor n' of n that satisfies the following three conditions:

(1) n' is a non-flexible iterative expression,

(2) there exists an a-labeled position in $First(n')$, and

(3) $SupLast(n) \preccurlyeq n'$.

Note that the a-labeled position may belong to $First(n)$. We can easily compute in linear time a pointer $NextNFlex(n, a)$ for all n of color a.

We observe that it is simultaneously possible that $NextNFlex(n, a) \neq Null$ and $Next(n, a) \neq Null$, even within deterministic expressions. To see this, consider the expression $e = ((aa)^{2..2})a$, with p_1, p_2, p_3 denoting the a-labeled positions from

left to right, and with the node n denoting the subexpression (aa), with witness p_2. Then $NextNFlex(n, a)$ is the parent of n, $Next(n, a) = p_3$ and yet e is deterministic.

We adapt Lemma 3.4 according to Proposition 3.12:

Lemma 3.15. An expression e with numeric occurrence indicators is non-deterministic iff one of the following four conditions is satisfied:

(1) (P1) is false,

(2) (P2) is false,

(3) there exist $a \in \Sigma$, a node $n \in N_{t_a}$ of color a, and a position q in $\{FirstPos(n, a), Next(n, a)\}$ such that $foll^{-1}(q) \cap foll^{-1}(Witness(n, a))$ contains at least one position, and

(4) there exist $a \in \Sigma$, a node $n \in N_{t_a}$ of color a such that $Last(NextNFlex(n, a)) \cap foll^{-1}(Witness(n, a))$ contains at least one position.

Proof. By definition, the conditions are obviously sufficient to guarantee non-determinism. We next show they are necessary. Let e be a non-deterministic expression that satisfies (P1) and (P2). According to Proposition 3.12 (2), there are necessarily positions z, x and y with $x \neq y$ and $lab(x) = lab(y)$ matching one of the next two options. Let x, y, z be as in Proposition 3.12 (1). Let $n = parent(SupFirst(x))$ and $n' = parent(SupFirst(y))$. According to Lemma 2.5, $n \preccurlyeq z$ and $n' \preccurlyeq z$. We assume w.l.o.g. that $n' \preccurlyeq n$. If $n \preccurlyeq y$ then $y \in FirstPos(n, a)$ because $n' \preccurlyeq n$. Otherwise, $y \in Next(n, a)$ because $y \in foll(z)$ and $n \preccurlyeq z$. Let now x, y, z be as in Proposition 3.12 (2), and let $n = parent(SupFirst(x))$. Then $NextNFlex(n, a) \neq Null$ and $SupLast(z) \preccurlyeq NextNFlex(n, a)$, which concludes the proof. □

3.4.3. Algorithm Testing Weak Determinism

In presence of numeric occurrences, it becomes more complicated to determine if Conditions (3) and (4) of Lemma 3.15 are satisfied, because we do not have an equivalent for Lemma 2.6. We therefore define a function $HighestFlex(n, n')$ which takes as input two nodes n and n' in e such that $n' \preccurlyeq n$, and returns the highest flexible iteration n'' such that $n' \preccurlyeq n'' \preccurlyeq n$ and $n' \neq n''$. If there is no such n'', then $HighestFlex(n, n') = Null$. For instance in $a(b(((c^{0..4})d)^{0..\infty}))$, if the nodes n and n' stand for the subexpressions c and $b(((c^{0..4})d)^{0..\infty})$, then $HighestFlex(n, n')$ is the node corresponding to the subexpression $((c^{0..4})d)^{0..\infty}$. Using techniques from [15], we can preprocess the parse tree of the expression in linear time so that each query $HighestFlex(n, n')$ can be answered in constant time:

Lemma 3.16. After a linear preprocessing of the expression e, each query $HighestFlex(n, n')$ can be answered in constant time.

Proof. We compute in a simple traversal of e a pointer from each node in e to its lowest ancestor that is a flexible iteration (or the root of the tree if there is no such

ancestor). Then we compute in linear time the skeleton t_{flex} of e, i.e., the tree whose nodes are the root of e plus all nodes of e that represent a flexible iteration. The tree t_{flex} is defined as an unranked tree, with ancestorship relations inherited from e: the number of children below each node in t_{flex} may be arbitrarily larger than 2, and may also obviously be smaller. Instead of left and right children, the siblings are ordered through a binary *next-sibling* relation in order to preserve the ordering of nodes (according to, say, pre-order traversal) from e.

We additionally keep a pointer $LC(x)$ from each node x of t_{flex} to the last (right-most) child of x in t_{flex}. We can view t_{flex} as a binary tree, since the "first-child/next-sibling" encoding B_{flex} of t_{flex} can be computed in linear time. In this encoding the first child of a node becomes the left child in the binary tree, and the next sibling of a node becomes the right child. We then index B_{flex} for *LCA* queries. As observed in Fact 9.1 of [15], $LCA_{B_{\text{flex}}}(LC(x), y)$ returns the child of x that is an ancestor of y in t_{flex}, for all nodes $x \preccurlyeq y$ in t_{flex}.

This allows us to compute $\mathit{HighestFlex}(n, n')$ in constant time: we follow the precomputed pointers to retrieve the lowest ancestors y (resp. x) of n (resp. n') that are flexible iterations. If $y = x$ then $\mathit{HighestFlex}(n, n') = \mathit{Null}$. Otherwise $\mathit{HighestFlex}(n, n')$ is obtained as $LCA_{B_{\text{flex}}}(LC(x), y)$. □

We finally provide a characterization of weak determinism that can be checked efficiently.

Theorem 3.17. An expression e with numeric occurrence indicators is not weakly deterministic if and only if it does not satisfy (P1) or (P2), or there exists $a \in \Sigma$ and a node n with color a such that one of the following conditions is satisfied:

(1) $\mathit{Next}(n, a)$, $\mathit{NextNFlex}(n, a)$ and $\mathit{NextFlex}(n)$ are not all equal to *Null*, and one of the following two conditions is satisfied

- $\mathit{Rchild}(n)$ is nullable or
- $\mathit{SupLast}(\mathit{HighestFlex}(\mathit{Witness}(n, a), n)) \preccurlyeq n$,

(2) or $\mathit{SupLast}(\mathit{HighestFlex}(\mathit{FirstPos}(n, a), n)) \preccurlyeq \mathit{Lchild}(n)$.

Proof. The claim follows from Lemma 3.15. We analyze below all possible cases that arise from the lemma and show each of them satisfies our claim.

1. $\mathit{foll}^{-1}(\mathit{Next}(n, a)) \cap \mathit{foll}^{-1}(\mathit{Witness}(n, a)) \neq \emptyset$ iff $\mathit{Next}(n, a) \neq \mathit{Null}$ and one of the following two conditions is satisfied:

   (A1) $\mathit{Rchild}(n)$ is nullable or

   (A2) $\mathit{SupLast}(\mathit{HighestFlex}(\mathit{Witness}(n, a), n)) \preccurlyeq n$.

2. $\mathit{Last}(\mathit{NextNFlex}(n, a)) \cap \mathit{foll}^{-1}(\mathit{Witness}(n, a)) \neq \emptyset$ iff $\mathit{NextNFlex}(n, a) \neq \mathit{Null}$ and one of the following conditions is satisfied:

   (B1) $\mathit{Rchild}(n)$ is nullable

(B2) $SupLast(HighestFlex(Witness(n,a),n)) \preccurlyeq n$.

3. $foll^{-1}(FirstPos(n,a)) \cap foll^{-1}(Witness(n,a)) \neq \emptyset$ iff $FirstPos(n,a) \neq Null$ and one of the following conditions is satisfied:

   (C1) $SupLast(HighestFlex(FirstPos(n,a),n))$ is an ancestor of $Lchild(n)$

   (C2) $NextFlex(n) \neq Null$ and one of the following two conditions is satisfied: (1) $Rchild(n)$ is nullable or (2) $SupLast(HighestFlex(Witness(n,a),n)) \preccurlyeq n$

We next prove the characterization. Actually, we only prove the "if" direction as the reverse implications are obvious.

1. Let $p \in foll^{-1}(Next(n,a)) \cap foll^{-1}(Witness(n,a))$. We deduce $n \preccurlyeq p$ from the fact that $p \in foll^{-1}(Witness(n,a))$ and $SupLast(p) \preccurlyeq n$ from $p \in foll^{-1}(Next(n,a))$. According to Lemma 3.14, if $Witness(n,a) \in foll^{\odot}(p)$ then $Rchild(n)$ is nullable because $SupLast(p) \preccurlyeq n$. Otherwise there exists some flexible expression x such that $p \in Last(x)$ and $Witness(n,a) \in First(x)$. Then, $SupLast(x) = SupLast(p) \preccurlyeq n$ which implies (A2).

2. The proof is exactly the same as for case 1, replacing $Next(n,a) \in foll(p)$ with $p \in Last(NextNFlex(n,a))$.

3. Let $p \in foll^{-1}(FirstPos(n,a)) \cap foll^{-1}(Witness(n,a)) \neq \emptyset$. We first observe that the position $FirstPos(n,a)$ is not in $foll^{\odot}(p)$. Otherwise $x = LCA(p, FirstPos(n,a))$ would necessarily be a descendant of $Lchild(n)$ because $Witness(n,a) \in foll(p)$ precludes $x \preccurlyeq n$. We deduce that x would be nullable since $SupLast(p) \preccurlyeq x$ and $SupFirst(FirstPos(n,a)) \preccurlyeq x$. But $SupLast(x) \preccurlyeq Lchild(n)$ and $SupFirst(x) \preccurlyeq n$ which would imply that $Lchild(n)$ is nullable, a contradiction.

   If $(p, FirstPos(n,a))$ belongs to $foll(Lchild(n))$, then we deduce from the preceding observation that (C1) is satisfied. Otherwise, $p \in foll^{-1}(Witness(n,a))$ implies that $n \preccurlyeq p$, and therefore $NextFlex(n,a) \neq Null$ and $SupLast(p) \preccurlyeq n$. In turn, $SupLast(p) \preccurlyeq n$ and $p \in foll^{-1}(Witness(n,a))$ implies that

   $$SupLast(HighestFlex(Witness(n,a),n)) \preccurlyeq n$$

   or $Rchild(n)$ is nullable. □

From this theorem we get immediately a linear algorithm to test determinism.

Theorem 3.18. Weak determinism of a regular expression e with numeric occurrence indicators can be tested in linear time $O(|e|)$, for an arbitrary alphabet.

3.4.4. Testing Strong Determinism

We next extend our algorithm to test strong determinism. Actually, most of the difficulty lies with testing weak determinism. The additional conditions that must be satisfied by strongly deterministic expressions can very easily be tested with our data structures.

Theorem 3.19. Strong determinism of a regular expression e with numeric occurrence indicators can be tested in linear time $O(|e|)$, for an arbitrary alphabet.

Proof. By Theorem 3.18 we can already assume that e is weakly deterministic. Therefore, we only need to check that for every pair of positions p and q, there is at most one way for q to follow p. Only two cases need to be considered:

(1) q follows p through a concatenation and through an iterative expression, and

(2) q follows p through two distinct iterative expressions

The first case occurs iff e contains a node n satisfying the following conditions: (1) $lab(n) = \odot$, (2) $SupLast(Lchild(n)) \preccurlyeq s$, and (3) $SupFirst(Rchild(n)) \preccurlyeq s$ where s is the closest iterative ancestor of n. The second case occurs iff e contains a node n satisfying the following three conditions: (1) $lab(n) = k..l$ with $k < l$, (2) $SupLast(Lchild(n)) \preccurlyeq s$, and (3) $SupFirst(Rchild(n)) \preccurlyeq s$ where s is the closest iterative ancestor of n. It is clear that one can check the existence of nodes satisfying those conditions in linear time. □

Gelade, Gyssens, and Martens [27] check essentially the same conditions (mixed with the verification that no pair of distinct positions sharing the same label can follow p). However, they base their algorithm on the follow relation instead of using pointers in the tree and so their algorithm has cubic time complexity. Their algorithm does not mention explicitly flexibility, but Algorithm 1 in [27] is only correct (modulo a typo in l.6 where first(s_1) should be first(s_2)) if weak determinism has been checked beforehand, which actually involves flexibility. This is because otherwise the algorithm does not identify correctly an expression like $e_5 = ((a^{2..3}+b)^2)^2 b$ as non-deterministic. The algorithm was improved by Chen and Lu [30] into a linear time test for strong determinism that again relies on flexibility to compute the follow relations.

4. Matching

In this section we present a collection of algorithms matching a word w against a *deterministic* expression e. First, we present an algorithm for arbitrary deterministic regular expressions that uses the constructions from Section 3 and lowest color ancestor queries to achieve expected time complexity $O(|e| + |w| \log\log |e|)$. Next, we present a matching algorithm for k-occurrence regular expressions in time $O(|e| + k|w|)$, which is linear if k is a constant. The most intricate matching algorithm that we present in this paper is the path-decomposition algorithm. It works in

time $O(|e| + c_e|w|)$, where c_e is the maximal depth of alternating union and concatenation operators in e. The three algorithms above perform matching by providing a *transition simulation procedure*: given a position p and a symbol a return the position q labeled a that follows p, or *Null* if no such position exists. If $e = (\#e')\$$, matching a word w against e' is straightforward: begin with position #, use the transition simulation procedure iteratively on subsequent symbols of w, and finally test if the position obtained after processing the last symbol of w is followed by \$.

The algorithms above allow to match multiple input words $w_1, \ldots, w_N$ against one regular expression e: the corresponding running times are obtained by replacing the factor $|w|$ by $|w_1| + \ldots + |w_N|$. We also present an algorithm that runs in time $O(|e| + |w_1| + \ldots + |w_N|)$ for star-free deterministic regular expressions e, a setting in which none of the previously mentioned algorithms guarantee linear complexity.

In the remainder of this section, we fix a deterministic regular expression e. Whenever we discuss positions and nodes, we implicitly mean positions and nodes of e.

4.1. Lowest Colored Ancestor Algorithm

Our previous construction that tests determinism in linear time provides an efficient procedure for transition simulation. Recall that we color the parent of any *IsSupFirst*-node n with the labels of the positions that belong to $First(n)$. By Lemma 3.3, given a position p and a symbol a, the a-labeled position q that follows p is one of: $Witness(n, a)$, $FirstPos(n, a)$, and $Next(n, a)$, where n is the lowest ancestor of p with color a. We use the *checkIfFollow* test (Theorem 2.4) to select the correct following position q among the three candidates.

Example 4.1. Consider the expression in Figure 4, position p_3, and the symbol c. The lowest ancestor of p_3 with color c is n_3. Here, $Witness(n_3, c) = p_5$, $Next(n_3, c) = p_1$, and $FirstPos(n_3, c) = Null$. Using *checkIfFollow* we find that it is p_5 that follows p_3. This ends the transition simulation procedure. Now, at position p_5 we read the next symbol a. The lowest ancestor of p_5 with color a is again n_3. This time it is $FirstPos(n_3, a) = p_2$ that follows p_5.

The basic ingredient of this procedure is an efficient algorithm for answering lowest colored ancestor queries. Recall the following result from Muthukrishnan et al., that will be discussed further at the end of this section:

Lemma 4.2 ([16, 17]). *Given a tree t with colors assigned to its nodes (some nodes possibly having multiple colors), we can preprocess t with a Las Vegas randomized algorithm in expected time $O(|t| + C)$, where C is the total number of color assignments, so that any lowest colored ancestor query is answered in time $O(\log \log |t|)$.*

Using Lemma 4.2, the transition simulation is accomplished in time $O(\log \log |e|)$, which gives us the following result.

Theorem 4.3. For any deterministic regular expression e, after preprocessing in expected time $O(|e|)$, we can decide for any word w whether $w \in L(e)$ in time $O(|w| \log\log |e|)$.

The nearest color ancestor algorithm is crucial for Theorem 4.3, and uses skeleta techniques similar to ours. We consequently briefly discuss this algorithm from [16, 17]. During preprocessing, the algorithm will:

- assign Euler tour numbers to each node of t
- build for each color c the tree R_c formed by the closure under LCA of all c-colored nodes
- attach to each node of R_c a pointer toward its nearest ancestor in R_c having color c (because nodes obtained through LCA closure do not have color c)
- preprocess R_c for LCA queries
- record in a Van Emde Boas (vEB) priority queue the c-colored nodes of t according to their Euler tour numbers

To answer a nearest ancestor query at node $n \in N_t$ with color c , we then:

- compute the predecessor and successor of n in the priority queue for color c.
- compute the LCA l of these nodes in R_c
- return the lowest ancestor of l with color c (using the pointer attached to l)

Our description diverges from the original algorithm [16, 17] in some minor (inconsequential) aspects: instead of building ancestor pointers through a linear traversal of R_c, the original algorithm attaches pointers to leaves, etc. The collection of all trees R_c (with the ancestor pointers in R_c) can be built in deterministic time $O(|t| + C)$. The vEB priority queues can be built in expected time $O(|t| + C)$ using a randomized Las Vegas algorithm, and the ancestor queries are answered in deterministic time $O(\log\log |t|)$.

Lemma 4.4 ([31, 32]). *We can preprocess k integers in the range $1..U$ in expected time $O(k)$ so that predecessor and successor queries are supported in deterministic time $O(\log\log U)$.*

Randomization is only used in the vEB tree to replace the arrays of the traditional vEB structure with (perfect) hash functions. If we consider memory allocation (without initialization) to come for free and use the lazy array technique described in Section 4.3 instead of hashing, the construction of the vEB tree in Lemma 4.4 becomes deterministic, and therefore also the nearest color ancestor datastructure and the matching algorithm of Theorem 4.3.

4.2. Bounded Occurrence Algorithm

A regular expressions e is called *k-occurrence* (k-ORE for short) if each symbol $a \in \Sigma$ occurs at most k times in e. While every regular expression is k-ORE for a sufficiently large k, Bex, Neven, Schwentick, and Vansummeren [33] report that the majority of regular expressions in real-life XML schemas are in fact 1-ORES. Given a position p and a symbol a, to find the following a-labeled position q we only need to perform the *checkIfFollow* test (Theorem 2.4) on all a-labeled positions in e, which are gathered into a designated list during preprocessing of e. Thus, transition simulation is performed in time $O(k)$, which proves the following result:

Theorem 4.5. For any deterministic k-ORE e, after preprocessing in time $O(|e|)$, we can decide for any word w whether $w \in L(e)$ in time $O(k|w|)$.

We note that an analogous technique can be used to match a word w against a non-deterministic k-ORE e: we maintain a set P of at most k positions and when reading symbol a we identify among the a-labeled positions those that follow any of the positions in P. Here, reading one symbol requires $O(k^2)$ time, and thus, the matching can be done in time $O(k^2|w|)$ after $O(|e|)$ preprocessing.

4.3. Path Decomposition Algorithm

Next, we describe an algorithm for matching a word w against a regular expression e in time $O(|e| + c_e|w|)$, where c_e is the maximal depth of alternating union and concatenation operators in e. As mentioned at the end of the Introduction, c_e is bounded by 4 in real-life DTDs [18].

First, we extend the function *FirstPos* from Section 3 to arbitrary nodes rather than skeleta nodes: for a node $n \in N_e$ and a symbol a, $\mathit{FirstPos}(n, a)$ returns the unique a-labeled position in $\mathit{First}(n)$ and *Null* if it does not exist. Queries of the form $\mathit{FirstPos}(n, a)$ can be answered in constant time after preprocessing in time $O(|e|)$, but since *FirstPos* is not used in the final algorithm, we omit the implementation details.

Climbing Algorithm. We first present a simple transition simulation procedure that uses *FirstPos* and later improve it to obtain the desired evaluation algorithm. Given a position p and a symbol a, it suffices to find an ancestor n of p such that $q = \mathit{FirstPos}(\mathit{Rchild}(n), a)$ follows p (tested with *checkIfFollow*). If such an ancestor does not exist, then p has no a-labeled following position. The soundness of this procedure follows from that of *checkIfFollow* and the completeness from Lemma 2.5. A naïve implementation seeks the ancestor in question by climbing up the parse tree starting from p, which yields $O(\mathit{depth}(e))$ time per transition simulation and overall $O(|e| + \mathit{depth}(e) \cdot |w|)$ time for matching.

Path Decomposition. Our algorithm speeds up climbing the path using jumps that follow precomputed pointers. The precomputed pointers lead to nodes where we store an aggregation of the values of *FirstPos* for several nodes skipped during the jump. The pointers are defined using the notion of path decomposition of the parse tree.

Recall that a *path decomposition* of a tree is a set of pairwise disjoint paths covering all nodes of the tree, and here, a path means a sequence of nodes $n_1, \ldots, n_k$ such that n_i is the parent of n_{i+1}. Note that a path decomposition of a tree can be specified by the set of the topmost nodes of the paths, which is how we define the path decomposition of e. A node y of e is the *topmost node* of a path if it is the root of e, or satisfies one of the following conditions:

(i) $\mathit{IsSupLast}(y)$,

(ii) $\mathit{IsSupFirst}(y)$,

(iii) y is the nullable right child of its parent, or

(iv) y is the right child of a +-labeled node.

For a position p we define $\mathit{top}(p)$ as the topmost node of the path of the left sibling of $\mathit{SupFirst}(p)$.

Example 4.6. Consider the regular expression presented in Figure 6 together with its path decomposition. For this expression, $c_e = 4$ because there are at most 4 alternations of union and concatenation operators on any path of the expression and, in particular, it is 4 on the path from p_1 to the root node. Note that $\mathit{top}(p_1) = n_3$ and $\mathit{top}(p_2) = n_1$.

We now define the function h, which is similar to *FirstPos* but defined for topmost nodes only: $h(n, a)$ points to the a-labeled position p with $n = \mathit{top}(p)$, i.e., we assign $h(\mathit{top}(p), \mathit{lab}(p)) = p$ for every position p. For instance, in the expression in Figure 6, $h(n_3, a) = p_1$ and $h(n_1, d) = p_2$.

There exists a subtle connection between h and *FirstPos*. If we consider a topmost node n, then the values of h assigned to n can be viewed as an aggregation of values of *FirstPos* of several nodes $n_1, \ldots, n_k$, which are gathered from around the path (but not from the path). The decomposition of e ensures that the aggregation is collision-free, i.e., if $\mathit{FirstPos}(n_i, a) \neq \mathit{Null}$ for some i, then $\mathit{FirstPos}(n_j, a) = \mathit{Null}$ for all $j \neq i$. Formally, we state this property as follows.

Lemma 4.7. For any two different positions p and p', if $\mathit{top}(p) = \mathit{top}(p')$, then p and p' have different labels.

Proof. Let y denote the lowest node in the path of $\mathit{top}(p)$ and let p_0 denote some position in $\mathit{Last}(y)$. We show that p follows p_0. By definition of $\mathit{top}(p)$, the left sibling of $\mathit{SupFirst}(p)$ is on the path between y and $\mathit{top}(p)$. Therefore, $\mathit{SupLast}(p_0) = \mathit{SupLast}(y)$ is an ancestor of the left sibling of $\mathit{SupFirst}(p)$ because there is no

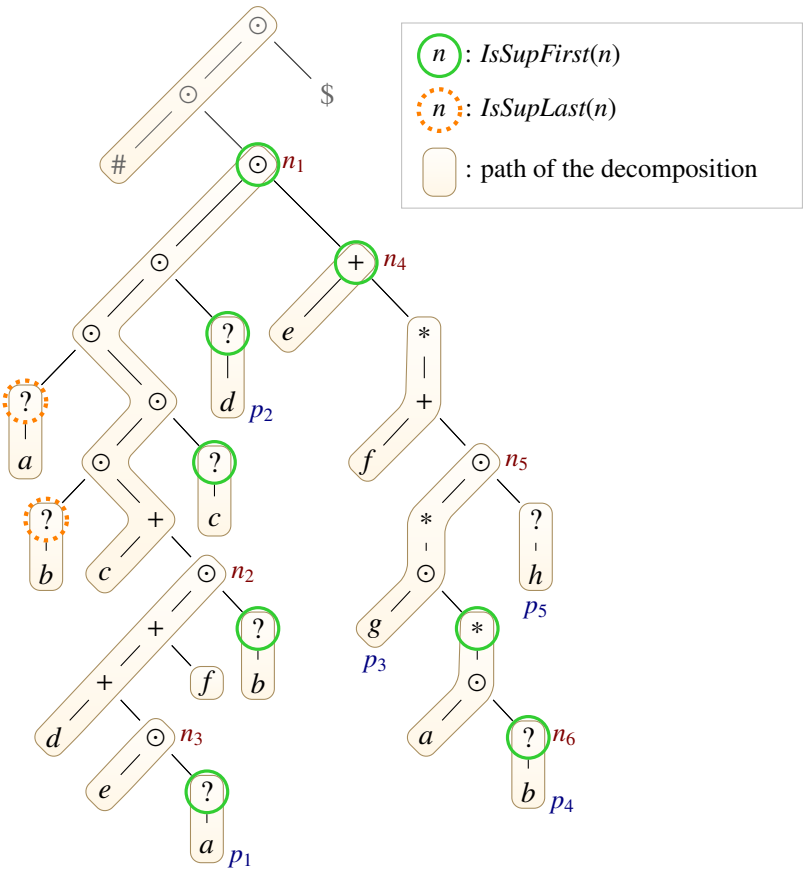

Figure 6: Path decomposition.

IsSupLast-node on a path except for the topmost node of the path. Moreover, we observe that the parent of $SupFirst(p)$ is labeled with $\odot$. Thus, by Lemma 2.2 we get $p \in Follow(p_0)$. Similarly, we show that $p' \in Follow(p_0)$. Because e is deterministic, there cannot be two different positions with the same label in $Follow(p_0)$. □

Lazy Arrays. To store the values of h we use *lazy arrays*, which we describe in detail next. This interesting data structure is known in programmer's circles [34, 35]; its first mention is, as far as we know, in Exercise 2.12 of [36]. The data structure provides the functionality of an associative array with constant time initialization, assignment, and lookup operations. The finite set of keys K needs to be known prior to initialization of the data structure. Furthermore, every key needs to be associated with a unique element from a continuous fragment of natural numbers, and here for simplicity, we assume that $K = \{1, \ldots, N\}$ for some $N \geq 1$.

A lazy array consists of an array A that stores the values associated with the

keys, a counter C of active keys having a value assigned, and additionally two arrays B and F that store the set of active keys. At initialization, C is set to 0 and uninitialized memory of length N is allocated for each of the arrays A, F, and B (an operation assumed to work in $O(1)$ time). To *assign* value v to key k, we add k to the set of active keys (if k is not in that set already), and assign $A[k] = v$. To *lookup* key k, we return $A[k]$ if k is active and return *Null* otherwise. To add a key k to the set of active keys, we increment C, set $F[C] = k$, and set $B[k] = C$. In this way a key k is active if and only if $1 \leq B[k] \leq C$ and $F[B[k]] = k$. Note that the first condition alone is insufficient to check if a key k is active because B has been allocated with uninitialized memory.

We found out that in practice, hash arrays offer compatible functionality with superior performance while theoretically providing only expected $O(1)$ time for the assignment and lookup operations. As a side note, we point out that lazy arrays stand on their own merit because they allow a constant time reset operation (by simply setting $C = 0$), unmatched by hash arrays (but not needed by our algorithm).

Preprocessing. We construct and fill the lazy array h in one bottom-up traversal of e. In the same traversal we also compute an additional pointer *nexttop* for every position and every topmost node of a path, defined as follows. We set $nexttop(n)$ to the lowest topmost node y of a path above $parent(n)$ that is either the root of e or satisfies one of the following conditions:

(1) $IsSupLast(y)$

(2) $IsSupFirst(y)$

(3) there exists a non-nullable $\odot$-labeled ancestor of n in the path of y.

For instance, in the expression in Figure 6, it holds that $nexttop(p_3) = n_5$, $nexttop(p_4) = n_6$, and $nexttop(p_5) = n_4$. We point out that $nexttop(n)$ is always the topmost node of some path, and furthermore, $nexttop(n)$ is a strict ancestor of n.

Transition Simulation. *FindNext* in Algorithm 4 follows *nexttop* pointers on the path from p to the node $SupLast(p)$ while attempting to find a-labeled follow positions stored in h at the visited nodes.

If this does not succeed, then *FindNext* checks in $First(parent(SupLast(p)))$ (Lines 8–14) to find follow positions. This task would be easy to accomplish with *FirstPos* through $FirstPos(parent(SupLast(p)), a)$. Since we wish to use h instead, we need to locate the node n such that $h(n, a)$ returns the position we look for. The location of this node depends on whether or not the node $y = parent(SupLast(p))$ is nullable. If y is nullable, we perform a single *nexttop* jump from y to reach n. Otherwise, n is the left sibling of y. Finally, we remark that if $FirstPos(parent(SupLast(p)), a)$ is not *Null*, then $h(n, a)$ returns the same node.

Example 4.8. Consider the expression in Figure 6, position p_1, and symbol d. The computation of $FindNext(p_1, d)$ follows the jump sequence: $p_1, parent(p_1), n_3, n_2, n_1$.

Algorithm 4: Transition simulation.

```
procedure FindNext(p : Position, a : Σ) : Position
1   x ← p
2   while SupLast(p) ≠ x
3       if checkIfFollow(h(x, a), p)
4           then return h(x, a)
5       x ← nexttop(x)
6   if checkIfFollow(h(x, a), p)
7       then return h(x, a)
8   y ← SupFirst(parent(x))
9   if y is nullable
10      then q ← h(nexttop(y), a)
11      else q ← h(Lchild(parent(y)), a)
12  if checkIfFollow(q, p)
13      then return q
14      else return Null
end procedure
```

At node n_1, $h(n_1, d)$ yields position p_2, and since p_2 follows p_1, the procedure returns p_2.

Correctness. To reason about iterations of the main loop of *FindNext*, we introduce this notation: $nexttop^0(n) = n$, and $nexttop^{i+1}(n) = nexttop(nexttop^i(n))$ for $i \geq 0$. Also, the *jump sequence* of p is the sequence

$$nexttop^0(p), nexttop^1(p), \ldots, nexttop^K(p),$$

where K is such that $nexttop^K(p) = SupLast(p)$. We call K the *length* of the jump sequence of p. We first show that the main loop performs a sufficient number of *nexttop* jumps.

Lemma 4.9. Let p be a position. For every position q that follows p, either $top(q)$ belongs to the jump sequence or q belongs to $First(parent(SupLast(p)))$.

Proof. By Lemma 2.5, $top(q)$ is an ancestor of p or the left sibling of a non-nullable *IsSupFirst*-ancestor of p. Furthermore, if $SupFirst(q)$ is nullable then $top(q)$ is the top of the path containing $parent(SupFirst(q))$. From the definition of *top* and *nexttop*, the jump sequence of p visits every *IsSupFirst*- and *IsSupLast*-ancestor of p, as well as every ancestor y of p such that y is the topmost node of a path and there exists some non-nullable $\odot$-labeled ancestor of p on that path.

We assume that $q \notin First(parent(SupLast(p)))$ (Assumption (A)) and show that in this case $top(q)$ belongs to the jump sequence. Let $n = LCA(p, q)$. We claim that $SupFirst(q)$ is the right sibling of $SupLast(p)$, or satisfies

$$SupLast(p) \preccurlyeq parent(SupFirst(q)) \preccurlyeq p.$$

By Lemma 2.2, one of the three following cases must hold:

(i) $lab(n) = \odot$, $q \in First(Rchild(n))$, $SupLast(p) = Lchild(n)$.

(ii) $lab(n) = \odot$, $q \in First(Rchild(n))$, $SupLast(p) \preccurlyeq n$.

(iii) $q \in First(s)$ and $p \in Last(s)$ where s is the lowest $*$-labeled ancestor of n.

If $q \notin First(n)$ case (iii) is obviously ruled out. and $SupFirst(q) = Rchild(n)$. The claim is then satisfied in both cases (i) and (ii). Otherwise we have $q \in First(n)$ and case (i) is ruled out by Assumption (A). We then have $SupLast(q) \preccurlyeq n$ and $SupFirst(q) \preccurlyeq n$ so that $SupLast(p) \preccurlyeq parent(SupFirst(q))$ by Assumption (A). This concludes the proof of our claim.

If $SupFirst(q)$ is the right sibling of $SupLast(p)$, then $top(q)$ is equal to $SupLast(p)$ and is therefore visited by the jump sequence. Otherwise, $SupLast(p) \preccurlyeq parent(SupFirst(q))$. By Lemma 2.6, $SupFirst(q)$ is nullable. Consequently, its parent belongs to the path of $top(q)$. Furthermore, the left sibling of $SupFirst(q)$, and therefore its parent, are non-nullable. It follows that the parent of $SupFirst(q)$ is a non-nullable $\odot$-labeled ancestor of p that belongs to the path of $top(q)$. Hence $top(q)$ belongs to the jump sequence. □

We now show the correctness of *FindNext*.

Lemma 4.10. For any position p and any symbol a, the procedure $FindNext(p, a)$ returns q iff $q \in Follow(p)$ and $lab(q) = a$.

Proof. The soundness of *FindNext* follows from the use of *checkIfFollow* prior to returning a position. Reciprocally, let $q \in Follow(p)$ with $lab(q) = a$. Then according to Lemma 4.9 *FindNext* returns q at Lines 4 or 7 if q does not belong to $First(parent(SupLast(p)))$, and at Line 13 otherwise. □

Complexity. We show that the amortized running time of the transition simulation procedure in Algorithm 4, when matching a word w against the deterministic regular expression e, is proportional to c_e, the maximal depth of alternating union and concatenation operators in e.

Lemma 4.11. Procedure $FindNext(p, a)$ works in amortized time $O(c_e)$, when matching a word against a deterministic regular expression e.

Proof. We use the potential *pot* of the data structure defined as a function of the current position:

$$pot(p) = |\{v \preccurlyeq p \mid IsSupFirst(v)\}|.$$

At the phantom position #, the initial potential is set to zero. The potential is increased by at most one each time the transition simulation procedure is executed. Now, let q be the position returned by $FindNext(p, a)$, i.e., the a-labeled position that follows p in e. We prove that $FindNext(p, a)$ executes at most $2(pot(q) - pot(p)) + c_e + O(1)$ iterations of the loop (*nexttop* jumps) before returning q.

By definition of *top*, there are no *IsSupFirst*-nodes between $SupFirst(q)$ and $top(q)$, hence

$$pot(q) \leq pot(top(q)) + 1. \tag{2}$$

Let K be the length of the jump sequence of p and let $n_i = nexttop^i(p)$ for $0 \leq i \leq K$. Now, from the sequence $n_0, \ldots, n_K$ we remove every node that is the non-nullable right child of a +-labeled node and obtain a subsequence $n_{i_0}, n_{i_1}, \ldots, n_{i_{K'}}$. We show that for every $1 \leq j \leq K' - 2$, one of n_j or n_{j+1} is an *IsSupFirst* node. If n_{i_j} is not an *IsSupFirst* node then by definition of the sequence it is the nullable right child of some $\odot$-labelled node, which is therefore also nullable. Then $n_{i_{j+1}}$ is an *IsSupFirst*-node by definition of *nexttop*. Hence, for every $0 \leq j \leq K'$,

$$j \leq 2(pot(n_{i_0}) - pot(n_{i_j})) + 2.$$

Thus, for every $0 \leq j \leq K$,

$$j \leq 2(pot(p) - pot(n_j)) + 2 + K - K'. \tag{3}$$

Let ℓ be the natural number such that $n_\ell = top(q)$. Combining equations (2) and (3), as c_e is an upper bound for $K - K'$, we obtain the result claimed before:

$$\ell \leq 2(pot(p) - pot(q)) + 4 + c_e. \tag{4}$$

From this result, establishing the amortized complexity is straightforward. Given a word $w = a_1 \cdots a_n$, let $p_1, \ldots, p_n$ be the sequence of positions with $p_i = FindNext(p_{i-1}, a_i)$ for $1 \leq i \leq n$ and $p_0 = \#$. Then, the number of iterations through the loop of *FindNext* while matching w against e is at most:

$$\begin{aligned} &n(4 + c_e) + 2\sum_{i=1}^{n}(pot(p_{i-1}) - pot(p_i)) \\ &= n(4 + c_e) + 2(pot(p_0) - pot(p_n)) \\ &\leq n(4 + c_e). \end{aligned}$$

This implies the amortized cost of $O(c_e)$, because each line of *FindNext* runs in constant time. □

Note that in the previous proof it suffices to take a smaller value of c_e, the maximum number of ancestors of a position of e that are labeled with +, are non-nullable, and have a parent labeled with $\odot$. Lemmas 4.10 and 4.11 thus respectively guarantee the correctness and analyze the complexity of our algorithm that simulates successive transitions. This proves the following result:

Theorem 4.12. For any deterministic regular expression e, after preprocessing in time $O(|e|)$, we can decide for any word w whether $w \in L(e)$ in time $O(c_e|w|)$, where c_e is the maximal depth of alternating union and concatenation operators in e.

4.4. Star-Free Algorithm

Finally, we present an algorithm that matches simultaneously several words $w_1, \ldots, w_N$ against a star-free deterministic regular expression e (i.e., one that contains no Kleene star). Note that the language defined by a star-free regular expression is finite. For a single word this is trivial: in a star-free regular expression, $q \in \mathit{Follow}(p)$ implies that position q is after p in the preorder traversal of e, and therefore, to simulate a transition it suffices to run the *checkIfFollow* test on subsequent positions until a match is found. In fact, the *checkIfFollow* tests can be hard-coded into the traversal to avoid lowest common ancestor queries. Obtaining linear complexity $O(|e|+|w_1|+\cdots+|w_n|)$ when matching several words $w_1, \ldots, w_N$ is much more challenging. We first outline our algorithm, then show how this outline can be implemented to achieve linear complexity.

Algorithm Outline. Our matching algorithm performs a single preorder traversal of the expression e, similarly to the algorithm above for the trivial single word problem. More accurately; after some preprocessing (which may involve more traversals), the algorithm iterates over the positions of e in a single pass from left to right. We maintain during this traversal:

- an index j_i for every word w_i
- a special data structure, which we call the *dynamic a-skeleton* for each letter $a \in \Sigma$.

The index j_i indicates the prefix of w_i matched so far during the preorder traversal of e: with every position visited in the traversal we increment the relevant indices $j_1, \ldots, j_N$. The dynamic a-skeleton introduced hereafter allow to determine efficiently which indices are to be incremented.

Before defining the skeleta, we first introduce some terminology. We say that the word w_i at index j_i *expects* the symbol a if the symbol of w_i at index $j_i + 1$ is a. We also say that w_i at j_i *reaches* position p if after simulating transitions on the corresponding prefix of w_i we arrive at p (or more precisely, the Glushkov automaton of e reaches p after reading the prefix of w_i). The *dynamic a-skeleton* t_a is a tree. At any point in the traversal, the leaves of this tree are the set of positions p such that there exists some w_i that reaches p and expects the symbol a at j_i. With each position p in t_a we associate a list of (pointers to) those words (i.e. p has a pointer to w_i iff w_i at index j_i reaches the position p and expects the symbol a). The internal nodes of the *dynamic a-skeleton* are obtained by closing the set of positions under lowest common ancestors.

During the preorder traversal of the positions in e, we update the dynamic skeleta as follows. When processing a position p labeled a, we remove from the dynamic a-skeleton t_a every position q that is followed by p, update indices of all words w_i on the list associated with q, and insert p into some dynamic skeleta. More accurately, if the letter in w_i at the new index j_i is a, then we insert p in the dynamic a-skeleton, with w_i attached to its list. We illustrate the procedure in the following example.

Example 4.13. We consider the deterministic regular expression

$$e = (\#(((a + ba)(c?))(d?b)))\$,$$

where # and \$ are two phantom positions that do not need to be matched. The expression e has 8 positions: $\#, a_1, \ldots, b_6, \$$. We match against e the words $w_1 = bcdb$, $w_2 = acdba$, $w_3 = acb$, and $w_4 = bada$. The evolution of skeleta through the algorithm is illustrated in Figure 7 and described below.

Initially, all indices are $j_1 = j_2 = j_3 = j_4 = 0$. When describing dynamic a-skeleta, we write $\langle p, W\rangle$ to indicate that a position p has an associated list of words W. Initially, $t_a = \langle \#, [w_2, w_3]\rangle$, $t_b = \langle \#, [w_1, w_4]\rangle$, and all other dynamic a-skeleta are empty.

In the first step, we read position a_1 (labeled a). Because a_1 follows #, we remove from t_a the position $\langle \#, [w_2, w_3]\rangle$, increment j_2 and j_3, and insert $\langle a_1, [w_2, w_3]\rangle$ to t_c.

Next, we read position b_2. Because b_2 follows #, we remove from t_b the position $\langle \#, [w_1, w_4]\rangle$, increment j_1 and j_4, and insert $\langle b_2, [w_4]\rangle$ to t_a and $\langle b_2, [w_1]\rangle$ to t_c. Because we keep the dynamic a-skeleta closed under lowest common ancestors, t_c becomes $\langle a_1, [w_2, w_3]\rangle + \langle b_2, [w_1]\rangle$, where + is a binary node whose children are a_1 and b_2.

At the position a_3, because a_3 follows b_2, we remove $\langle b_2, [w_4]\rangle$ from t_a, increment j_4 and add $\langle a_3, [w_4]\rangle$ to t_d. At the position c_4 labeled with c, because c_4 follows a_1, we remove from t_c the position $\langle a_1, [w_2, w_3]\rangle$, increment j_2 and j_3, and insert $\langle c_4, [w_2]\rangle$ to t_d and $\langle c_4, [w_3]\rangle$ to t_b. Although b_2 is not followed by c_4, we also remove $\langle b_2, [w_1]\rangle$ from t_c and discard it because we observe that b_2 will not be followed by any of the subsequent positions. After this step, $t_b = \langle c_4, [w_3]\rangle$, t_c is empty, and $t_d = \langle a_3, [w_4]\rangle \odot \langle c_4, [w_2]\rangle$.

The next position, d_5, follows both a_3 and c_4. Therefore, we remove from t_d both $\langle a_3, [w_4]\rangle$ and $\langle c_4, [w_2]\rangle$, increment j_2 and j_4, and insert $\langle d_5, [w_4]\rangle$ to t_a and $\langle d_5, [w_2]\rangle$ to t_b. This way, t_b is $\langle c_4, [w_3]\rangle \odot \langle d_5, [w_2]\rangle$.

In the last step we move to the position b_6 labeled with b. Because b_6 follows both c_4 and d_5, we remove $\langle d_5, [w_2]\rangle$ and $\langle c_4, [w_3]\rangle$ from t_b and increment j_2 and j_3. We insert $\langle b_6, [w_2]\rangle$ to t_a. Because $j_3 = |w_3|$ and \$ follows b_6, w_3 matches e. Since there are no further positions to process, the words w_1, w_2, and w_4 do not match d.

Implementation Details. Details on how to efficiently handle dynamic skeleta follow. We assume that the positions $p_1, \ldots, p_m$ of e are given in the traversal order of e and that e has been preprocessed for *LCA* and ancestor queries, and that pointers *SupLast* and *SupFirst* are computed. This preprocessing can be performed in time $O(|e|)$.

In every dynamic a-skeleton t_a we assume we can access in constant time the rightmost position p_a, i.e., the position most recently added to t_a (for this we can maintain a pointer to p_a). Our algorithm will also rely on a procedure

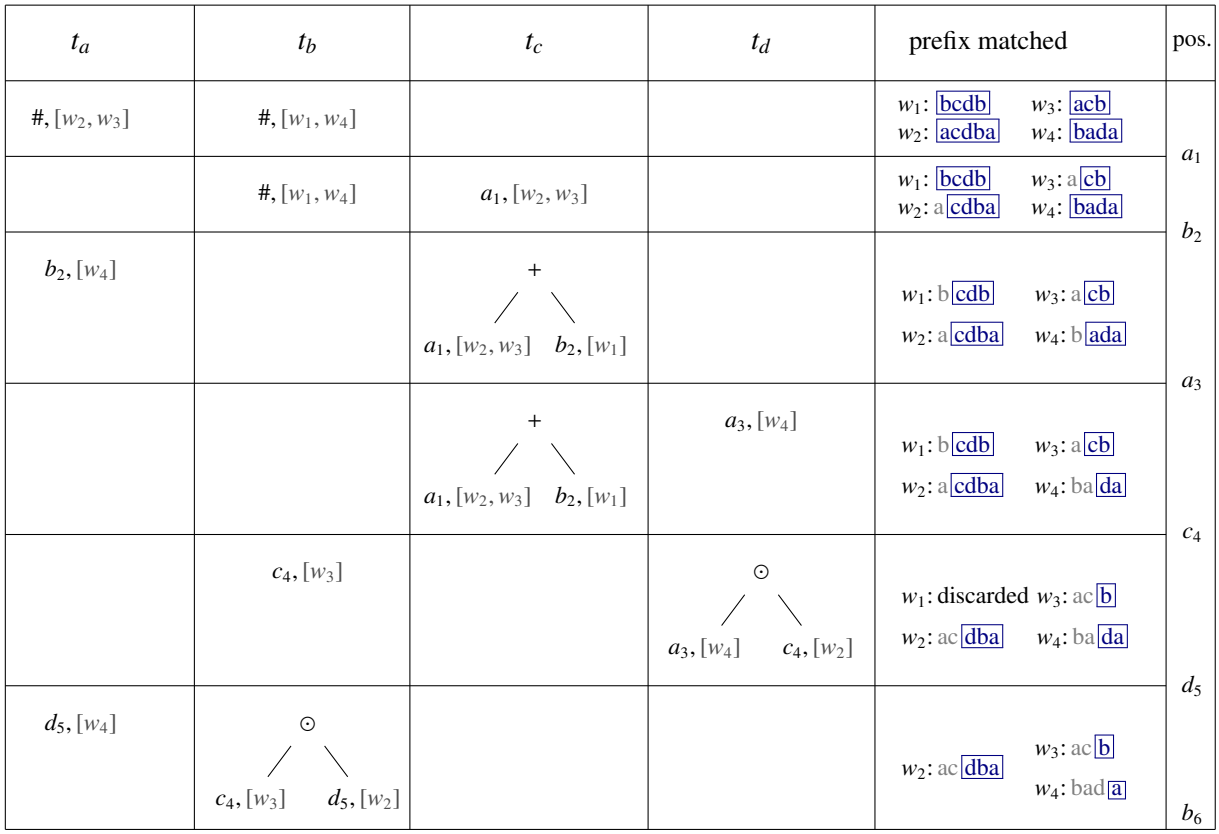

Figure 7: Dynamic skeleta while matching $w_1, \ldots, w_4$ against $e = (\#(((a + ba)(c?))(d?b)))\$$

findLCA(t_a, p_i) to insert new positions as well as to identify and to remove relevant positions from the dynamic a-skeleta. The *findLCA* procedure returns the location in t_a of the lowest common ancestor n_{LCA} between p_a and p_i. Note that n_{LCA} needs not be present in t_a and *findLCA*(t_a, p_i) returns the topmost descendant of n_{LCA} present in t_a (which may be n_{LCA} itself if t_a contains it).

Our implementation takes advantage of two assumptions on the calls to *findLCA* by our algorithm:

(i) *findLCA*(t_a, p_i) is only called with a position p_i that comes after p_a in the traversal of e (there exists some $j < i$ such that $p_a = p_j$)

(ii) positions involved in successive calls to *findLCA* form an increasing subsequence of $p_1, \ldots, p_m$ (if a call *findLCA*(t_a, p_i) is followed by a call *findLCA*(t_a, p_j), then $i < j$).

Due to (i) we can implement *findLCA* by simply climbing the rightmost path in t_a until the desired node is found. Due to (ii) we can save the result of the previous call and begin to climb the rightmost path of t_a from the saved node (if no new nodes have been added in between). We will see below that this implementation reduces the amortized cost of a call to *findLCA* in our algorithm to $O(1)$.

When processing a position p_i labeled a, our matching algorithm first find the lowest common ancestor n_{LCA} of n_a and p_i in e. Then we use $\mathit{findLCA}(t_a, p_i)$ to find the location of n_{LCA} in t_a (Step 1). Using this location, we first retrieve all leaves $\langle p, L\rangle$ in t_a such that p is a position followed by p_i. Then for every word w_h in L, if we denote by b the letter at index j_h in w_h:

- we insert p_i in the dynamic b-skeleton t_b (unless of course p_i has already been inserted for another word w'_h in which case it would already be in t_b so insertion would be redundant)
- we attach w_h to p_i in the dynamic b-skeleton t_b (i.e., append w_h to the list associated with p_i in t_b)
- we increment j_h

We name this set of operations "Step 2". Finally, we remove from t_a nodes that have become irrelevant after inserting p_i (Step 3).

To perform the retrieval and removal operations in Steps 2 and 3, we climb the path from $\mathit{findLCA}(t_a, p_i)$ to $n_i = \mathit{parent}(\mathit{SupFirst}(p_i))$. At every $\odot$-labeled node n along this path, we pick the subtree t' rooted at the left child of n. In a single traversal of t' we collect every position p of t': if $\mathit{SupLast}(p) \preccurlyeq \mathit{Lchild}_e(n_{LCA})$, then p is followed by p_i according to Lemma 2.2 (so we perform the retrieval steps described above). Otherwise, the words associated with p fail to match the expression and can be discarded (this is because p cannot be followed by any position $p_h, h > i$ as our left to right order implies $\mathit{LCA}(p, p_h) \preccurlyeq \mathit{LCA}(p, p_i)$). After all positions in t' have been processed, we discard the subtree t' from t_a. This concludes the description of our implementation. We next analyze its running time.

Complexity Analysis. First, we observe that for every consumed word symbol (i.e., every increment of a word counter), we add into dynamic skeleta a constant amount of nodes and words: at most one position, one additional word in the list of a position, and one additional *LCA* node into some dynamic skeleton, then remove the position to which the word ws attached. Consequently, at any time, there is in the whole data structure (i.e. over the whole set of dynamic skeleta) at most one pointer to every word. Our second observation is that the number of calls to Procedure *findLCA* is $|e|$ in Step 1 and $|w_1| + \cdots + |w_N|$ in Step 2. As a consequence, the cumulative execution time of *findLCA* is bounded by $O(|w_1| + \cdots + |w_N| + |e|)$ over the whole execution of the algorithm: the nodes traversed by climbing the rightmost path in procedure *findLCA* leave the rightmost path and therefore the algorithm may only traverse $O(|w_1| + \cdots + |w_N|)$ such nodes in total. The costs of the retrieval/removal/insertion operations add up to $O(|w_1| + \cdots + |w_N|)$, which justifies the following theorem:

Theorem 4.14. For any star-free deterministic regular expression e and words $w_1, \ldots, w_N$, we can decide which words belong to $L(e)$ in time $O(|e| + |w_1| + \cdots + |w_N|)$.

5. Matching Strongly Deterministic Regular Expressions

Consider an expression e with numerical occurrence indicators. For each node n of e we define $NextIter(n)$ as a pointer to the closest iterative node among the strict ancestors of n (and to $Null$ otherwise). We now adapt our three matching algorithms from the previous section to support numeric occurrence indicators, using pointer $NextIter$ to examine and update the counters in relevant iterative subexpressions. The main difficulty is to maintain the efficiency while additionally traversing the pointers $NextIter$. Our matching algorithms in presence of numeric occurrence indicators rely again on an algorithm that tests if a configuration can follow another one.

Algorithm 5 presents a general procedure to check if some position follows another one, given a valuation c and the lowest iterative ancestor x of the current position for which $c(x) < min(x)$. We recall from Section 2.2 that $c(x)$ denotes the value of the counter attached to x and $min(x)$ the minimal number of repetitions required by subexpression x while matching e – i.e., the numeric occurrence indicators over x are $x^{min(x)..max(x)}$. To simplify algorithms using this procedure, we require from the procedure that it also returns the iterative expression corresponding to the transition if any, and $\odot$ otherwise.

Proposition 5.1. Let $y_{<min}$ be the lowest $y \preccurlyeq p$ for which $c(y) < min(y)$, and $y_{<min}$ is $Null$ if there is no such y. Then Algorithm 5 returns $false$ on $(p, q, c, y_{<min})$ if and only if there are no s, c' such that $(q, c') \in Follow^s(p, c)$, returns $(true, \odot)$ if and only if there exists c' such that $(q, c') \in Follow^{\odot}(p, c)$, and otherwise returns $(true, s)$ where s is the lowest iterative expression such that $(q, c') \in Follow^s(p, c)$.

Proof. It is clear that the algorithm checks the characterization above. □

Algorithm 5: Checking a candidate position

procedure *checkIfFollow*(p, q : Positions, c : valuation, $y_{<min}$) : Bool× Iterative expr.

```
1  n ← LCA(p, q)
2  if y_<min ≠ Null and n ≼ y_<min then return (false, Null)
3  if lab(n) = ⊙ and SupLast(p) ≼ n and SupFirst(q) ≼ n then return (true, ⊙)
4  s ← NextIter(n)
5  while s ≠ Null and SupLast(p) ≼ s and SupFirst(q) ≼ s and max(s) = c(s)
6      s ← NextIter(s)
7  if s = Null or max(s) = c(s)
8      then return (false, Null)
9      else return (true, s)
```

end procedure

5.1. Bounded Occurrence Expressions

We describe an efficient algorithm for k-ORES. Assume that e is a k-ORE (as defined in Section 4.2). We first introduce some data structure that helps us process iterative subexpressions efficiently in our algorithm.

Stacks and Pointers. We store two stacks in addition to the counters. The first, *StackMin*, keeps a pointer to the counter of every ancestor n of the current position such that $c(n) < min(n)$. The second stack $Stack_{c\geq 2}$ keeps a pointer to every ancestor n of the current position such that $c(n) \geq 2$ (these are the counters that may be reset during the execution of the algorithm). The stack operations in Algorithm 6 guarantee that both stacks are ordered by decreasing depth of the nodes, so that the deepest nodes come on top. During preprocessing, we also build for each node n a pointer $NextIter(n)$ toward the closest iterative node among the strict ancestors of n, and another pointer $NextIter_{min\geq 2}(n)$ toward the lowest iterative node among the strict ancestors of n such that $min(n) \geq 2$.

Algorithm Outline. We gather for each letter $a \in \Sigma$ all a-labeled positions of e into a designated list during preprocessing, following the preorder. Our matching algorithm then starts from the initial configuration (#, counters set to 1). It processes the symbols of the input word in order and updates the configuration as follows. When reading letter a from position p, we first reorder the list of a-labeled positions into $q_1 \dots q_k$ such that q comes before q' when $LCA(p, q')$ is a strict ancestor of $LCA(p, q)$. This reordering can clearly be obtained with $O(k)$ operations. We then execute procedure *FindNextPosK* to compute the next configuration. This procedure relies on the procedure *checkIfFollow* to check successively the candidate positions q_i, and then updates the counters using *StackMin* and $Stack_{c\geq 2}$.

In the end, either the procedure fails at some point to produce a following configuration, or the sequence of configurations produced testifies that e matches the input word according to Lemma 2.7. The correctness of the algorithm is guaranteed by Proposition 5.1: using Lemma 2.7 we can show that the counters and the two stacks are correctly updated. More interesting is the complexity of the algorithm.

Complexity Analysis. Given a valuation c for the counters and a node n, we define the first potential of n as $pot_c(n) = |\{x \preccurlyeq nc(x) \geq 2\}|$. The iterations of the while loop inside *checkIfFollow* visit distinct nodes for every candidate position q_i. More importantly, each visited node n satisfies $c(n) = max(n) \geq 2$. As the candidates are ordered by ascending order of $LCA(p, q_i)$, the counter of every node thus visited is reset in the subsequent loops (Line 12). Furthermore, at most one counter is incremented while simulating the transitions (Line 16). Consequently, the overall complexity of the calls to procedure *checkIfFollow* is bounded by $C_1(k + pot_c(p) - pot_{c'}(q_i) + 2)$ for some constant C_1, where p and c (resp. q_i and c') are the current state and valuation before (resp. after) simulating the transition. Similarly, the number of operations for updating $Stack_{c\geq 2}$ (Lines 13,17,18) can be bounded by $C_2(pot_c(p) - pot_{c'}(q_i) + 2)$ for some constant C_2.

Algorithm 6: Transition Simulation for k-ORE

```
//uses: StackMin, Stack_{c≥2} : Stacks of iterative nodes, c : valuation
procedure FindNextPosK(p, q_1, q_2 ..., q_k : Positions, a : Σ) : Position
1   i ← 0; b ← false
2   while i < k and b = false
3       i ← i + 1
4       (b, x) ← checkIfFollow(p, q_i, c, top(StackMin))
5   if b = false then return false
6   else
7       UpdateCounters(p, q_i, x)
8       return q_i

procedure UpdateCounters(p, q_i, x):
9   L_1 ← empty stack
10  s ← x
11  while Stack_{c≥2} is not empty and top(Stack_{c≥2}) ⋠ s
12      c(top(Stack_{c≥2})) ← 1
13      pop(Stack_{c≥2})
14  if x ≠ ⊙
15  then
16      c(x) ← c(x) + 1
17      if top(Stack_{c≥2}) ≠ x
18          then push(x, Stack_{c≥2})
19      if c(x) = min(x)
20          then pop(StackMin)                    //removes x from StackMin
21   z ← NextIter_{min≥2}(q_i)                    } updates StackMin
22  while z ≠ Null and z ⋠ s                      }
23      push(z, L_1)                              }
24      z ← NextIter_{min≥2}(z)                   }
25  while L_1 is not empty                        }
26      z ← pop(L_1)                              }
27      push(z, StackMin)                         }
end procedure
```

Let us define the second potential of a node n as $pot'_c(n) = |\{x \preccurlyeq nc(x) < min(x)\}|$. We observe that when a node is inserted into *StackMin*, its counter has value 1 and its minimal value is at least 2 so that it contributes to the second potential for all its descendants. Furthermore, the second potential can decrease by at most one during a transition simulation since procedure *checkIfFollow* guarantees that nodes contributing to the second potential of p are above $LCA(p, q_i)$, and even above x if $x \neq Null$. Thus, the number of operations for updating *StackMin* (Lines 21 to 27) can be bounded by $C_3(pot'_{c'}(q_i) - pot'_c(p) + 2)$ for some constant C_3.

For a large enough constant C, this gives an overall complexity of $C(pot_c(p) - pot_{c'}(q_i) + pot'_{c'}(q_i) - pot'_c(p) + 4)$ for the cost of one transition simulation. From this result, establishing the amortized complexity is straightforward. Given a word $w = a_1 \cdots a_n$, let $p_1, \ldots, p_n$ be the sequence of positions with $p_i = FindNextPosK(p_{i-1}, a_i)$ and $c_1, \ldots, c_n$ the corresponding valuations, for $1 \leq i \leq n$ and $p_0 = \#$. The results before show that for a large enough constant C, the number of operations while matching w against e is at most:

$$
\begin{aligned}
&Ckn + 4Cn + C\sum_{i=0}^{n-1}(pot_{c_i}(p_i) - pot_{c_{i+1}}(p_{i+1})) + C\sum_{i=0}^{n-1}(pot'_{c_{i+1}}(p_{i+1}) - pot'_{c_i}(p_i)) \\
&= Ckn + 4Cn + C(pot_{c_0}(p_0) - pot_{c_n}(p_n) + pot'_{c_n}(p_n) - pot'_{c_0}(p_0)) \\
&\leq Ckn + 6Cn \in O(kn).
\end{aligned}
$$

We have thus established that deterministic k-ORE expressions can still be evaluated in linear time (for fixed k) even in presence of counters:

Theorem 5.2. For every strongly deterministic k-ORE with numerical occurrence indicators, after preprocessing in time $O(|e|)$, we can decide if $w \in L(e)$ in time $O(k|w|)$.

5.2. Lowest Color Ancestor

Instead of reordering the list of a-labeled positions into $q_1 \cdots q_k$ prior to verifying the certificates, we can can traverse the a-colored ancestors of p using lowest color ancestor queries in time $O(\log\log|e|)$, and take advantage of pointers $FirstPos(n, a)$ and $Witness(n, a)$ to check these positions in the order specified in Section 5.1, i.e., such that q comes before q' when $LCA(p, q')$ is a strict ancestor of $LCA(p, q)$. Lemma 3.3 does not carry over in presence of numeric occurrence indicators, but it is obvious that the next position q_i is still among $FirstPos(n, a)$ or $Witness(n, a)$ for some a-colored ancestor n of p. To visit the q_i in desired order, we can consider the candidates from $\bigcup_{n \preccurlyeq p \text{ with color } a}\{Witness(n, a), FirstPos(n, a)\}$ by height of n. We compute in constant time lowest common ancestors of p with $FirstPos(n, a)$ and $Witness(n, a)$ in order to decide which of these two positions should be considered first. One proves easily that with this approach the evaluation of deterministic expressions with counters has cost $O(|w|\log\log|e|)$ after the preprocessing:

Theorem 5.3. For every strongly deterministic expression with counters, after preprocessing in expected time $O(|e|)$, we can decide if $w \in L(e)$ in time $O(|w| \log\log |e|)$.

5.3. Path Decomposition

Instead of resorting to lowest color ancestor queries, we can use the path decomposition idea to enumerate the candidates that need to be checked. The path decomposition algorithm for standard expressions enumerates the candidates q_i in increasing order of $parent(SupFirst(q_i))$. To restore the proper order for our numeric indicators algorithms (namely, q comes before q' when $LCA(p, q')$ is a strict ancestor of $LCA(p, q)$), we combine this enumeration with pointers $FirstPos(n, a)$ as for the lowest color ancestor algorithm. We slightly alter the definition of the *nexttop* pointers to preserve the property that all relevant ancestors are traversed by the jump sequence (a property whose proof depends on Lemma 2.6 in the absence of numeric occurrence indicators). For every node n such that n is not nullable and n is an *IsSupFirst*-node, $nexttop(n)$ is redefined as the left sibling of n. The transition simulation procedure must then be modified accordingly, so that the jump sequence only concludes on node n if the two following conditions are satisfied simultaneously: $IsSupLast(n)$ and $n \preccurlyeq p$. We can easily prove that after those modifications all relevant candidates are visited.

Theorem 5.4. For any deterministic expression e with numeric occurrence indicators, after preprocessing in time $O(|e|)$, we can decide for any word w whether $w \in L(e)$ in time $O(c_e|w|)$, where c_e is the maximal depth of alternating union and concatenation operators in e.

Proof. To prove the correctness we only need to show that the jump sequence in $FindNext(p_i, a)$ with the alterations described above traverses $top(q)$ for every a-colored ancestor n of p_i such that $SupLast(p_i)$ is a strict ancestor of n, and every $q \in \{FirstPos(n, a), Witness(n, a)\}$ such that $q \notin First(parent(SupLast(p_i)))$. The only case that differs from the proof of Lemma 4.9 is when $SupLast(p_i) \preccurlyeq parent(SupFirst(q_i))$. Then, if $SupFirst(q_i)$ is nullable the argument follows as in Lemma 4.9, but $SupFirst(q_i)$ may also be non-nullable (and still be a candidate in our list). If $SupFirst(q_i)$ is non-nullable, then its left sibling is an *IsSupLast* node, and therefore is visited by our modified jump sequence, which concludes our proof.

For the complexity analysis, we observe that some *IsSupFirst* ancestor of p_i is visited every three iterations, instead of two for the jump sequence analyzed in Lemma 4.11. As a consequence, the total cost over all transition simulations remains $O(c_e|w|)$ since our ordering of the candidates guarantees the total cost of processing counters is in $O(|w|)$. □

6. Related Work

Our results rely on preprocessing and traversing the parse tree of a regular expression. Definitions and constructions of the *First* and *Last* sets based on the

parse tree already appeared in works by Chan and Paige [11], Ponty, Ziadi, and Champarnaud [12], and Hagenah and Muscholl [26].

The idea of our algorithm for star-free regular expressions is similar to that of Hagenah and Muscholl [37] in their algorithm that computes for any regular expression an ε-free NFA in time $\Omega(|e| \log^2 |e|)$. They decompose the transitions leaving each state into a few sets and group states sharing such sets of outgoing transitions. This decomposition is based on a *heavy path* decomposition of the parse tree of e. We use another decomposition of this parse tree in order to amortize the evaluation cost.

Czerwiński, David, Losemann, and Martens [38] prove that checking determinism of NFAs and regular expressions is PSPACE-complete. Lu, Bremer, and Chen [39] give an alternative proof of this result, and also prove that for DFAs with limited alphabet size, checking determinism is NL-complete. Lu, Peng, Chen, and Zheng [40] show that checking determinism of regular expressions over a unary alphabet is coNP-complete.

Gelade, Gyssens, and Martens [27] present a cubic time algorithm for testing strong determinism of regular expressions with numerical occurrence indicators. They also show that strongly deterministic expressions are strictly less expressive than weakly deterministic expressions. Their algorithm is improved by Chen and Lu [30] to a linear time algorithm. For weak determinism, Kilpeläinen and Tuhkanen [22] present a quadratic testing algorithm, for expressions over a fixed alphabet. Kilpeläinen [29] improves this result to linear time in the fixed alphabet case, and to time $O(|e|^2 \log |e|)$ in the arbitrary alphabet case. Chen and Lu [30] improve that result to time $O(|\Sigma||e|)$, where Σ is the set of symbols that appear in e. It is shown by Latte and Niewerth [41] that a weakly deterministic regular expression is at most exponentially larger than an equivalent DFA, which implies that testing if a given regular expression is definable by a weakly deterministic expression can be decided in EXPSPACE.

An orthogonal direction of research involves algorithms for the efficient validation of huge documents against a small DTD. Several works [42, 43] focused on obtaining space efficient algorithms in a streaming framework. This is challenging when document trees are deep. Konrad and Magniez [44] provide streaming algorithms in sublinear space for the validation against DTDs. They consider a framework where the algorithm has access to a read-only input stream and several auxiliary read/write streams. The algorithm is allowed to perform read or write passes on the streams. At the beginning of each pass on a stream, the algorithm decides in which direction the stream is processed, and also decides if the pass is a write or a read pass. The authors propose an algorithm that validates a tree t against a constant-size DTD in $O(\log^2 |t|)$ passes, using space $O(\log |t|)$ and 3 auxiliary streams, with $O(\log |t|)$ processing time per symbol. Note that the validator checks the sibling sequences of t against the corresponding deterministic regular expression.

In the context of DTD inference, Bex et al. [45] identify two classes of regular expressions which account for most of the regular expressions in real schemas: the

single occurrence regular expressions (1-ORE) and the chain regular expressions (CHARE). An expression is an 1-ORE iff no symbol appears more than once in e, therefore 1-ORE are always deterministic. CHARE are a subclass of 1-ORE, and contain the 1-ORE that consist of a sequence of factors of the form $(a_1 + a_2 + \cdots + a_n)$ where every a_i is a symbol, each factor being possibly extended with a Kleene star, Kleene plus, or a question mark. The class of 1-ORES account for 98% of the regular expressions in real schemas, while CHARE account for 90% of them. Bex, Neven, and van den Bussche [46] also define *simple regular expressions*, which generalize CHARE in that symbols a_i in factors can appear with a star or question mark, and the number of occurrences of a symbol is not restricted. It should be noted that deterministic simple regular expressions have bounded depth of alternating union and concatenation operators, and thus is covered by our linear time algorithm for that class. Moreover although stars are allowed in simple regular expressions, which makes them unfit for our star-free algorithm, those stars can occur only above a single symbol, or above a union of strings (with possibly a star or question mark above the strings). Therefore, our star-free algorithm can also be extended to handle simple deterministic regular expressions.

7. Conclusions

We have presented a linear time algorithm for testing if a regular expression is deterministic, an efficient algorithm for matching words against deterministic regular expressions, and linear time algorithms for matching against k-occurrence, $*$-free (multiple words), and bounded $+$-depth expressions.

It was our original motivation for this work, but remains an open theoretical problem, whether matching for deterministic regular expressions can be carried out in time $O(|e| + |w|)$. We note that our $O(|e| + |w| \log\log|e|)$ matching algorithm is not optimal because of the $O(\log\log|e|)$ cost of lowest color ancestor queries. We plan to find out if the cost of those lowest colored ancestor queries can be amortized and if the particular order of the queries can be used to devise better data structures. Can other approaches solve the problem in $O(|e| + |w|)$ time, e.g., by giving up the the streaming aspect of using transition simulation? Which larger classes of regular expressions, exceeding the deterministic ones, can be matched efficiently? An example of such class is mentioned after Theorem 4.5, the k-ORES.

Another interesting and largely open problem is the one of matching under linear time preprocessing of w ("indexing"). Very simple matching problems such as substring search have time $O(|e|)$ solutions; can those be extended to more general regular expressions? The only related reference in this direction that we are aware of is the work by Baeza-Yates and Gonnet [47]. Finally, can lower bounds matching the upper bounds be shown? Note that for general regular expressions and NFAS, it is known that no approach relying on constructing an equivalent epsilon-free NFA can achieve linear complexity. This follows from the fact that all epsilon-free NFAS equivalent to $a_1?a_2? \ldots a_m?$ have at least $m \log^2 m$ transitions [48].

Finally we show that our linear algorithm for testing determinism extends to regular expressions with numeric occurrence indicators. Both weak and strong determinism can therefore be decided in linear time. Our matching algorithms also extend to strongly deterministic expressions in presence of numeric occurrence indicators, but we leave the complexity of matching for weakly deterministic expressions for future work.

Acknowledgments. We are very grateful to Sławek Staworko for his contributions to this project, and to the reviewers for their help to improve the presentation of this article.

References

[1] S. Kleene, Representation of events in nerve nets and finite automata, in: C. Shannon, J. Mccarthy (Eds.), Automata Studies, Princeton University Press, Princeton, N.J., 1956, pp. 3–42. (Cited page 1)

[2] K. Thompson, Regular expression search algorithm, Commun. ACM 11 (6) (1968) 419–422. (Cited page 1)

[3] P. Kumar, V. Singh, Efficient regular expression pattern matching for network intrusion detection systems using modified word-based automata, in: SIN, 2012, pp. 103–110. (Cited page 1)

[4] R. Cox, Regular expression matching in the wild, available at swtch.com/ rsc/regexp/regexp3.html (2010). (Cited page 1)

[5] C. F. Goldfarb (Ed.), Information processing–Text and Office Systems–Standard Generalized Markup Language (SGML), International Standard ISO 8879, International Organization for Standardization, Geneva, 1986. (Cited page 2)

[6] T. Bray, J. Paoli, C. M. Sperberg-McQueen, E. Maler (Eds.), Extensible Markup Language (XML) 1.0 (Second Edition), W3C Recommendation, 2000. (Cited page 2)

[7] D. C. Fallside, P. Walmsley (Eds.), XML Schema Part 0: Primer Second Edition, W3C Recommendation, 2004. (Cited pages 2 and 6)

[8] A. Brüggemann-Klein, D. Wood, One-unambiguous regular languages, Inf. Comput. 142 (2) (1998) 182–206. (Cited page 2)

[9] V. M. Glushkov, The abstract theory of automata, Russian Mathematical Surveys 16 (1961) 1–53. (Cited page 2)

[10] A. Brüggemann-Klein, Regular expressions into finite automata, Theor. Comput. Sci. 120 (2) (1993) 197–213. (Cited page 3)

[11] C.-H. Chang, R. Paige, From regular expressions to DFA's using compressed NFA's, TCS 178 (1–2) (1997) 1–36. (Cited pages 5, 8, 9, and 46)

[12] J.-L. Ponty, D. Ziadi, J.-M. Champarnaud, A new quadratic algorithm to convert a regular expression into an automaton, in: Workshop on Implementing Automata, 1996, pp. 109–119. (Cited pages 5, 8, 9, and 46)

[13] D. Harel, R. E. Tarjan, Fast algorithms for finding nearest common ancestors, SIAM J. Comput. 13 (2) (1984) 338–355. (Cited pages 5 and 9)

[14] M. A. Bender, M. Farach-Colton, G. Pemmasani, S. Skiena, P. Sumazin, Lowest common ancestors in trees and directed acyclic graphs, J. Algorithms 57 (2) (2005) 75–94. (Cited pages 5 and 9)

[15] M. Bojańczyk, P. Parys, XPath evaluation in linear time, J. ACM 58 (4) (2011) 17. (Cited pages 5, 15, 19, 20, 24, and 25)

[16] S. Muthukrishnan, M. Müller, Time and space efficient method-lookup for object-oriented programs, in: SODA, 1996, pp. 42–51. (Cited pages 5, 28, and 29)

[17] M. Farach, S. Muthukrishnan, Optimal parallel dictionary matching and compression, in: SPAA, 1995, pp. 244–253. (Cited pages 5, 28, and 29)

[18] S. Grijzenhout, Quality of the XML web, Master's thesis, University of Amsterdam, draft, see http://data.politicalmashup.nl/xmlweb/ (July 2010). (Cited pages 5 and 30)

[19] L. Wall, R. Schwartz, Programming Perl, O'Reilly and Associates, 1991. (Cited page 6)

[20] P. Hazel, Perl Compatible Regular Expressions, Cambridge University Press, 2003. (Cited page 6)

[21] H. Björklund, W. Martens, T. Timm, Efficient incremental evaluation of succinct regular expressions, in: CIKM, 2015, pp. 1541–1550. (Cited page 6)

[22] P. Kilpeläinen, R. Tuhkanen, One-unambiguity of regular expressions with numeric occurrence indicators, Inf. Comput. 205 (6) (2007) 890–916. (Cited pages 6, 11, 12, 20, 21, 22, and 46)

[23] B. Groz, S. Maneth, S. Staworko, Deterministic regular expressions in linear time, in: PODS, 2012, pp. 49–60. (Cited page 6)

[24] J. E. Hopcroft, J. D. Ullman, Introduction to Automata Theory, Languages and Computation, Addison-Wesley, 1979. (Cited page 6)

[25] A. Brüggemann-Klein, Regular expressions into finite automata, Theor. Comput. Sci. 120 (2) (1993) 197–213. (Cited page 7)

[26] C. Hagenah, A. Muscholl, Computing epsilon-free NFA from regular expressions in $O(n \log^2(n))$ time, in: MFCS, 1998, pp. 277–285. (Cited pages 9 and 46)

[27] W. Gelade, M. Gyssens, W. Martens, Regular expressions with counting: Weak versus strong determinism, SIAM J. Comput. 41 (1) (2012) 160–190. (Cited pages 12, 14, 27, and 46)

[28] C. Koch, S. Scherzinger, Attribute grammars for scalable query processing on xml streams., VLDB J. 16 (3) (2007) 317–342. (Cited page 14)

[29] P. Kilpeläinen, Checking determinism of XML schema content models in optimal time, Inf. Syst. 36 (3) (2011) 596–617. (Cited pages 21, 22, and 46)

[30] H. Chen, P. Lu, Checking determinism of regular expressions with counting, Inf. Comput. 241 (2015) 302–320. (Cited pages 27 and 46)

[31] P. van Emde Boas, Preserving order in a forest in less than logarithmic time and linear space, Inf. Process. Lett. 6 (3) (1977) 80–82. (Cited page 29)

[32] P. van Emde Boas, R. Kaas, E. Zijlstra, Design and implementation of an efficient priority queue, Mathematical Systems Theory 10 (1977) 99–127. (Cited page 29)

[33] G. J. Bex, F. Neven, T. Schwentick, S. Vansummeren, Inference of concise regular expressions and DTDs, TODS 35 (2). (Cited page 30)

[34] R. Kecher, Reset an array in constant time, Blog entry cplusplus.co.il (May 2009). (Cited page 32)

[35] B. Moret, H. Shapiro, Algorithms from P to NP, Benjamin/Cummings, 1990. (Cited page 32)

[36] A. V. Aho, J. E. Hopcroft, J. D. Ullman, The Design and Analysis of Computer Algorithms, Addison-Wesley, 1974. (Cited page 32)

[37] C. Hagenah, A. Muscholl, Computing epsilon-free NFA from regular expressions in o(n $\log^2$(n)) time, ITA 34 (4) (2000) 257–278. (Cited page 46)

[38] W. Czerwinski, C. David, K. Losemann, W. Martens, Deciding definability by deterministic regular expressions, in: FOSSACS, 2013, pp. 289–304. (Cited page 46)

[39] P. Lu, J. Bremer, H. Chen, Deciding determinism of regular languages, Theory Comput. Syst. 57 (1) (2015) 97–139. (Cited page 46)

[40] P. Lu, F. Peng, H. Chen, L. Zheng, Deciding determinism of unary languages, Inf. Comput. 245 (2015) 181–196. (Cited page 46)

[41] M. Latte, M. Niewerth, Definability by weakly deterministic regular expressions with counters is decidable, in: MFCS, 2015, pp. 369–381. (Cited page 46)

[42] L. Segoufin, C. Sirangelo, Constant-memory validation of streaming XML documents against DTDs, in: ICDT, 2007, pp. 299–313. (Cited page 46)

[43] L. Segoufin, V. Vianu, Validating streaming XML documents, in: PODS, 2002, pp. 53–64. (Cited page 46)

[44] C. Konrad, F. Magniez, Validating XML documents in the streaming model with external memory, in: ICDT, 2012. (Cited page 46)

[45] G. J. Bex, F. Neven, T. Schwentick, S. Vansummeren, Inference of concise regular expressions and DTDs, ACM Trans. Database Syst. 35 (2). (Cited page 46)

[46] G. J. Bex, F. Neven, J. V. den Bussche, DTDs versus XML Schema: A practical study, in: WebDB, 2004, pp. 79–84. (Cited page 47)

[47] R. A. Baeza-Yates, G. H. Gonnet, Fast text searching for regular expressions or automaton searching on tries, J. ACM 43 (6) (1996) 915–936. (Cited page 47)

[48] G. Schnitger, Regular expressions and NFAs without *epsilon*-transitions, in: STACS, 2006, pp. 432–443. (Cited page 47)