Journal of Computer and System Sciences **62**, 565–588 (2001)
doi:10.1006/jcss.2001.1748, available online at http://www.idealibrary.com on IDEAL®

Translating Regular Expressions into Small ϵ-Free Nondeterministic Finite Automata

Juraj Hromkovič[1] and Sebastian Seibert[1]

Lehrstuhl für Informatik I, RWTH Aachen, 52056 Aachen, Germany
E-mail: jh@cs.rwth-aachen.de, seibert@cs.rwth-aachen.de

and

Thomas Wilke

Institut für Informatik und Praktische Mathematik, Christian-Albrechts-Universität zu Kiel, 24098 Kiel, Germany
E-mail: wilke@ti.informatik.uni-kiel.de

Received April 5, 1999; revised December 14, 2000

We prove that every regular expression of size n can be converted into an equivalent nondeterministic ε-free finite automaton (NFA) with $\mathcal{O}(n(\log n)^2)$ transitions in time $\mathcal{O}(n^2 \log n)$. The best previously known conversions result in NFAs of worst-case size $\Theta(n^2)$. We complement our result by proving an almost matching lower bound. We exhibit a sequence of regular expressions of size $\mathcal{O}(n)$ and show the number of transitions required in equivalent NFAs is $\Omega(n \log n)$. This also proves there does not exist a linear-size conversion from regular expressions to NFAs. © 2001 Academic Press

Key Words: regular expressions; finite automata.

1. INTRODUCTION

One of the central tasks of formal language theory is to describe infinite objects (languages) by finite formalisms (automata, grammars, expressions, etc.), and to investigate the descriptional power and complexity of these formalisms. In this paper, we consider two standard formalisms for describing regular languages: nondeterministic finite automata without ε-transitions (NFAs) and regular expressions. The size (descriptional complexity) of an NFA is considered to be the number of its transitions; likewise, the size of a regular expression is considered to be the number of occurrences of alphabet symbols in it.

[1] Supported by the Deutsche Forschungsgemeinschaft under Project HR 14/3-2.

0022-0000/01 $35.00
Copyright © 2001 by Academic Press
All rights of reproduction in any form reserved.

On the one hand, it is known, see [5], that the conversion of NFAs into equivalent regular expressions may lead to a considerable increase of the descriptional complexity; i.e., there are regular languages requiring regular expressions of size exponential in the size of their minimal NFAs. On the other hand, previously described conversions from regular expressions into NFAs (see, e.g., [1, 8, 11, 13]) produce automata with worst-case size quadratic in the size of the input. In [12],[2] it is even claimed that the sequence of regular languages defined by $(a_1+\varepsilon)(a_2+\varepsilon)\cdots(a_n+\varepsilon)$, for each n, requires NFAs of size $\Omega(n^2)$, which would imply that the above conversions are optimal.

In this paper, we devise a polynomial-time conversion procedure from regular expressions to NFAs that produces automata of size $\mathcal{O}(n(\log n)^2)$ where n denotes the size of the input. This is an essential improvement over the previously known conversions and disproves the lower bound claimed in [12]. We show that our construction is almost optimal by proving a lower bound of $\Omega(n \log n)$ for the above-mentioned example from [12]. This also implies the nonexistence of linear-size conversions from regular expressions to NFAs.

The starting point of our construction is what we call the "position automaton" for a regular expression. This automaton, first described in [1] and also known as "nondeterministic Glushkov automaton" [2], is based on ideas already explained in [10] and [6].

The basic idea of our conversion is very simple. We first replace each state in the position automaton by a few copies—each copy gets the same incoming edges as the original state, and the outgoing edges of the original state are distributed among its copies—and then identify states that have the same outgoing edges. (Obviously, we obtain an equivalent automaton.) The number of transitions increases in the first step, while it decreases in the second step. So the crucial point is to distribute the outgoing edges in the first step in such a way that identifying states in the second step leads to an overall smaller number of transitions. See Fig. 1 for an illustration of this idea.

At the heart of our conversion algorithm there is a recursive procedure that is invoked with a subset of the set of states of the position automaton as argument. It determines how to distribute the outgoing edges of the states in this set among their copies (and, of course, how many copies of each state should be generated). This is done by partitioning the set into two smaller subsets and applying the procedure recursively. The partitioning process is guided by the structure of the parse tree of the given regular expression. Each state in the position automaton corresponds to a leaf of this tree; to determine a useful partition of the given set the procedure analyzes the subtree of the parse tree induced by the leaves corresponding to the given set of states. (Note that this subtree will, in general, *not* correspond to a subexpression of the given regular expressions. Thus our procedure cannot easily be turned into a procedure that works by induction on the structure of the regular expression itself.)

Efficient algorithms for acceptance testing and DFA construction from regular expressions are of great interest to various applied areas such as string matching and lexical scanning, and there is an extensive body of research that is concerned with the issues that arise, see, e.g., the survey article [3] on pattern matching. As

[2] See p. 77, and Exercise 3.7 on p. 106.

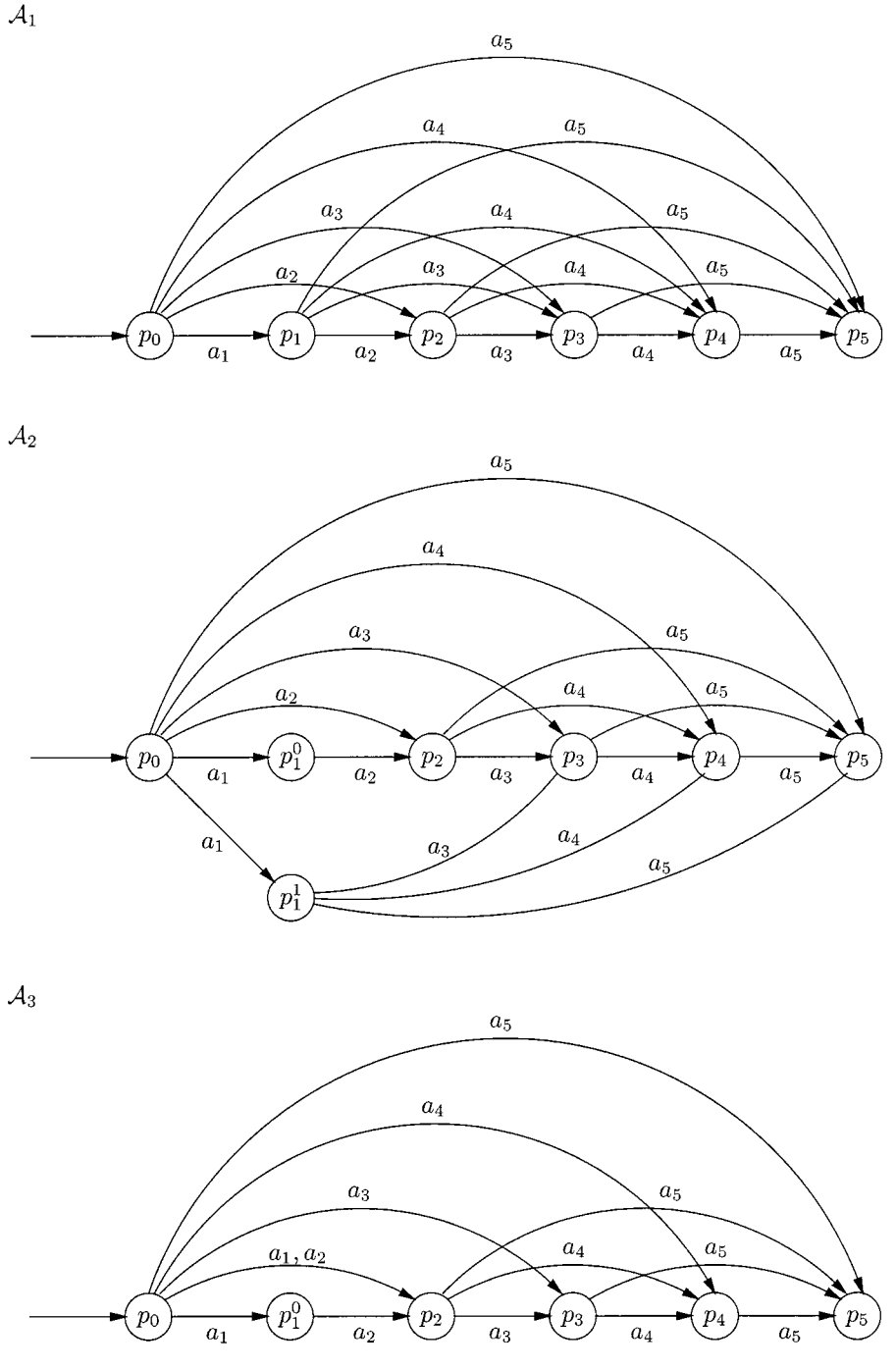

FIG. 1. Illustration of the basic idea. Automaton $\mathscr{A}_1$ is the position automaton for $(a_1+\varepsilon)\cdots(a_5+\varepsilon)$. Automaton $\mathscr{A}_2$ is obtained from the $\mathscr{A}_1$ by splitting p_1 into two states, p_1^0 and p_1^1, while $\mathscr{A}_3$ results from $\mathscr{A}_2$ by identifying p_1^1 and p_2. The number of transitions increases from 15 to 16 and is then reduced to 13. At the end of Section 7, we will demonstrate exactly how our algorithm handles this example.

most of the suggested solutions involve computing an equivalent NFA (or DFA) for the input regular expression our lower bound can be regarded as a hint to what can be hoped for when automata-theoretic methods are to be applied. It should, however, be noted that more advanced techniques (such as those presented and discussed in [4]) do not work with NFAs directly but compressed representations of NFAs, which our lower bound as such might not apply to.

This paper is an improved and extended version of the conference presentation [9].

2. BASIC TERMINOLOGY AND MAIN RESULT

When we speak of a *regular expression* over an alphabet $\mathbb{A}$, we mean a finite expression built from the symbols in $\mathbb{A}$ and the special symbols "$\varnothing$" and "ε" using the binary operation symbols "$+$" and "$\cdot$" and the unary operation symbol "*". Parentheses are used to indicate grouping, the operators "$+$" and "$\cdot$" are written in infix notation, "*" is written in postfix notation, and "$\cdot$" is often omitted. Given a regular expression E over an alphabet $\mathbb{A}$, we write $\mathscr{L}(E)$ for the subset of $\mathbb{A}^*$ that is denoted by E. The *size* of a regular expression E, denoted size (E), is the number of occurrences of elements from $\mathbb{A}$ in E.

When we speak of a *nondeterministic finite automaton* (NFA) over an alphabet $\mathbb{A}$, we mean a tuple (Q, q_I, Δ, Q_F) where Q is a finite set of states, q_I is the initial state, $\Delta \subseteq Q \times \mathbb{A} \times Q$ is the finite transition relation, and Q_F is the set of final states. We thus do not allow ε-transitions in NFAs! Given an NFA $\mathscr{A}$, we denote by $\mathscr{L}(\mathscr{A})$ the subset of $\mathbb{A}^*$ recognized by $\mathscr{A}$.

THEOREM 2.1. *There is a procedure that*

(a) *converts every regular expression E of size n into an equivalent NFA with at most $2n-1$ states and at most*

$$\frac{4}{(\log_2 3/2)^2} n(\log_2 n)^2$$

transitions and

(b) *runs in time $\mathcal{O}(n^2 \log n)$.*

3. POSITION AUTOMATA

In this section, we explain what we mean by the "position automaton" of a regular expression and introduce notation and terminology that comes with this. We follow in part the exposition in [2].[3]

We assume that every regular expression E over an alphabet $\mathbb{A}$ comes with a set of *positions*, denoted pos(E), whose elements are in one-to-one correspondence with the occurrences of letters from $\mathbb{A}$ in E. A position can be best thought of as pointing to a particular occurrence of a letter in E. One can for instance enumerate the positions by $\{1, 2, ..., n\}$, and let position i point the ith occurrence of a letter if there are n occurrences of letters in n.

[3] In [1], slightly different definitions for the "first" and "last" set of a regular expression are given, which are less useful for our purposes than the definitions from [2].

Given a regular expression E and a position $x \in \text{pos}(E)$, we write $\langle E \rangle_x$ for the letter in E that x points to (i.e., occurs in position x).

Let E be a regular expression over an alphabet $\mathbb{A}$. When scanning a word from $\mathscr{L}(E)$, each letter scanned matches a particular occurrence of this letter in E, or—in our terminology—corresponds to a particular position of E. Of course, there is often more than just one way to scan a word from $\mathscr{L}(E)$ (due to ambiguities of E). The sets first(E) and last(E) are defined in a way such that a position x belongs to first(E) (respectively last(E)) if and only if it corresponds to the first (respectively, last) letter scanned in some scanning process.

Formally, first(E) is defined by induction according to the following rules. Here, as when defining last and follow below, we always assume that F and G denote subexpressions of E such that the positions sets of F and G are disjoint:

$$\text{first}(\varnothing) = \varnothing, \tag{1}$$

$$\text{first}(\varepsilon) = \varnothing, \tag{2}$$

$$\text{first}(a) = \text{pos}(a) \qquad \text{for} \quad a \in \mathbb{A}, \tag{3}$$

$$\text{first}(F + G) = \text{first}(F) \cup \text{first}(G), \tag{4}$$

$$\text{first}(FG) = \begin{cases} \text{first}(F) & \text{if} \quad \varepsilon \notin \mathscr{L}(F), \\ \text{first}(F) \cup \text{first}(G) & \text{if} \quad \varepsilon \in \mathscr{L}(F), \end{cases} \tag{5}$$

$$\text{first}(F^*) = \text{first}(F). \tag{6}$$

In order to obtain rules for last(E) substitute "last" for "first" and replace (5) by

$$\text{last}(FG) = \begin{cases} \text{last}(G) & \text{if} \quad \varepsilon \notin \mathscr{L}(G), \\ \text{last}(F) \cup \text{last}(G) & \text{if} \quad \varepsilon \in \mathscr{L}(G). \end{cases} \tag{7}$$

Given a position $x \in \text{pos}(E)$, the set follow(E, x) is defined in a way such that $y \in \text{follow}(E, x)$ if and only if x is immediately followed by y in some scanning process. It is defined inductively according to the rules

$$\text{follow}(a, x) = \varnothing \qquad \text{for} \quad a \in \mathbb{A}, \tag{8}$$

$$\text{follow}(F + G, x) = \begin{cases} \text{follow}(F, x) & \text{if} \quad x \in \text{pos}(F), \\ \text{follow}(G, x) & \text{if} \quad x \in \text{pos}(G), \end{cases} \tag{9}$$

$$\text{follow}(FG, x) = \begin{cases} \text{follow}(F, x) & \text{if} \quad x \in \text{pos}(F) \backslash \text{last}(F), \\ \text{follow}(F, x) \cup \text{first}(G) & \text{if} \quad x \in \text{last}(F), \\ \text{follow}(G, x) & \text{if} \quad x \in \text{pos}(G), \end{cases} \tag{10}$$

$$\text{follow}(F^*, x) = \begin{cases} \text{follow(F, x)} & \text{if} \quad x \in \text{pos}(F) \backslash \text{last}(F), \\ \text{follow}(F, x) \cup \text{first}(F) & \text{if} \quad x \in \text{last}(F). \end{cases} \tag{11}$$

The first, last, and follow sets of a regular expression E can be used to define an NFA that recognizes $\mathscr{L}(E)$ in the following way, as was already described in [1].

We define the *position automaton* for E, denoted $\mathscr{A}_E$, to be the quadruple (Q, q_I, Δ, Q_F) where, for some $q_I \notin \text{pos}(E)$,

$$Q = \{q_I\} \cup \text{pos}(E),$$

$$Q_F = \begin{cases} \{q_I\} \cup \text{last}(E) & \text{if } \varepsilon \in \mathscr{L}(E), \\ \text{last}(E) & \text{otherwise,} \end{cases}$$

$$\Delta = \{(q_I, \langle E \rangle_x, x) \mid x \in \text{first}(E)\} \cup \{(x, \langle E \rangle_y, y) \mid y \in \text{follow}(E, x)\}.$$

LEMMA 3.1. *For every regular expression E, $\mathscr{L}(E) = \mathscr{L}(\mathscr{A}_E)$.*

In Section 4, we define for each regular expression E a variety of NFAs—the "common follow sets automata" for E—all of which recognize E. This is our starting point. The following sections deal with the problem of finding (i.e., constructing) a particularly small "common follow sets automaton" for each regular expression. After having introduced further terminology in Section 5, we present a first, "basic" solution in Section 6, which is then improved in Section 7. The proof that our constructions work correctly is given subsequently in Section 8, while in Section 9 it is shown that our constructions do achieve the claimed upper bounds on the size of the resulting NFAs. Section 10 explains how our procedure can be implemented so as to run efficiently, and we conclude with a lower bound in Section 11.

4. COMMON FOLLOW SETS AUTOMATA

The idea behind our construction is to decompose the follow set of each position, i.e., the set of successors of a state of the position automaton, into some subsets. These subsets become the states of a new automaton, each subset C being responsible for the transitions from the original state to the elements of C. When the position automaton is in a state x (a position of E), the new automaton will be in one of the chosen subsets of follow(E, x) instead, say in C. Each transition from x to every $x' \in C$ is replaced by transitions from C to every C' belonging to the decomposition of follow(E, x'). As the same set C can occur in several decompositions of different follow sets—we think of it as being used "commonly"—this potentially leads to an overall reduced number of transitions!

What we have just explained is only a rough sketch; the definition below is a little more complicated due to the fact that sharing a set C might interfere with marking a state as final or nonfinal.

DEFINITION 4.1. Let E be a regular expression, given with its set of positions pos(E).

A *system of common follow sets* (*system of CFS*) for E is given by a *decomposition* $\text{dec}(x) \subseteq \mathscr{P}(\text{pos}(E))$ for every $x \in \text{pos}(E)$ and is required to satisfy $\text{dec}(x) \neq \varnothing$ and

$$\text{follow}(E, x) = \bigcup_{C \in \text{dec}(x)} C$$

for every $x \in \text{pos}(E)$.

The *family of common follow sets* (*family of CFS*) $\mathscr{C}$ associated with this system is defined by

$$\mathscr{C} = \{\text{first}(E)\} \cup \bigcup_{x \in \text{pos}(E)} \text{dec}(x).$$

The *common follow sets automaton* (*CFS automaton*) associated with this system is the tuple (Q, q_I, Δ, Q_F) defined by

$$Q = \mathscr{C} \times \{0, 1\};$$

$$q_I = \begin{cases} (\text{first}(E), 1) & \text{if } \varepsilon \in \mathscr{L}(E), \\ (\text{first}(E), 0) & \text{otherwise;} \end{cases}$$

$$\Delta = \{((C, f), \langle E \rangle_x, (C', f_x)) \mid x \in C, f \in \{0, 1\}, C' \in \text{dec}(x)\},$$

$$Q_F = \mathscr{C} \times \{1\},$$

where $f_x = 1$ if $x \in \text{last}(E)$ and $f_x = 0$ if $x \notin \text{last}(E)$, for $x \in \text{pos}(E)$.

LEMMA 4.1. *Let E be a regular expression and $\mathscr{A}$ the CFS automaton associated with any system of CFS for E. Then $\mathscr{L}(E) = \mathscr{L}(\mathscr{A})$.*

Proof. We must show that $w \in \mathscr{L}(E)$ iff $w \in \mathscr{L}(\mathscr{A})$ for any word w. For $w = \varepsilon$, this is trivial. For $w \neq \varepsilon$, we use Lemma 3.1. Following Lemma 3.1, it is sufficient to show that the following conditions (i) and (ii) are equivalent for every w:

(i) There exists a position x such that, after having read w, the position automaton $\mathscr{A}_E$ can be in state x.

(ii) There exists a position x such that, after having read w, the CFS automaton $\mathscr{A}$ can be in state (C, f_x) where $C \in \text{dec}(x)$.

We prove this fact by induction on $|w|$

For the induction base, assume $|w| = 1$, say $w = a$ for some letter a. We only have to expand the definitions of $\mathscr{A}_E$ and $\mathscr{A}$. By definition of the position automaton, $\mathscr{A}_E$ goes from q_I to some state x when reading letter a if and only if $x \in \text{first}(E)$ and $a = \langle E \rangle_x$. By definition of the CFS automaton $\mathscr{A}$, there exists an a-labeled transition from the initial state to some state (C, f) if and only if there exists $x \in \text{first}(E)$ with $a = \langle E \rangle_x$, and $C \in \text{dec}(x)$. Moreover, $f = f_x$ in this case, which means $f = 1$ if and only if x is final in $\mathscr{A}_E$. This proves the claim for $|w| = 1$.

Assume w_0 is a nonempty string and $w = w_0 a$ for some letter a. Then $\mathscr{A}_E$ goes from q_I to some state x when reading $w_0 a$ if and only if it goes from q_I to some state x_0 when reading w_0 and $x \in \text{follow}(E, x_0)$ and $a = \langle E \rangle_x$. By induction hypothesis, this is the case if and only if $\mathscr{A}$ goes from q_I to some state (C_0, f_0) with $x_0 \in C_0$ when reading w and (just as before) $x \in \text{follow}(E, x_0)$ and $a = \langle E \rangle_x$. By definition of $\mathscr{A}$, this, in turn, is the case if and only if $\mathscr{A}$ goes from q_I to some state (C_0, f_0) with $x_0 \in C_0$ when reading w (just as before) and there exists a transition $((C_0, f_0), \langle E \rangle_x, (C, f_x))$ in $\mathscr{A}$. This is clearly equivalent to (ii). ∎

So far, we have seen that every CFS automaton for a given regular expression recognizes the language denoted by that expression, regardless of what system of CFS one starts from. However, the complexity of a particular CFS automaton obviously depends on the particular system of CFS it is built from. We want to find systems of CFS that yield small automata.

5. EXPRESSIONS AS TREES

From now on, we fix a regular expression E and a decomposition of E into subexpressions (i.e., we resolve ambiguities arising from iterated products and sums).

We represent E as a tree, which is denoted by t_E. In this tree, each node corresponds to exactly one occurrence of a subexpression of E. We identify the node and the subexpression. So, when we speak of a subexpression of E we really mean an occurrence of a subexpression.

Let F and G be subexpressions of E. If F is an ancestor of G (and thus contains G as a subexpression), we write $F \prec G$. The notion of a *subtree* will be understood in the graph theoretic manner, i.e., as a subgraph being a tree. As a special case, we consider *the subtree of t below F* as the tree consisting of a node F and all its descendants $F' \geqslant F$ in t. (This is what is sometimes called a subtree, whereas our notion of subtree allows, for instance, a single path of the original tree to be a subtree.) In particular, the entire subtree of t_E below some node F is the tree representation t_F.

For any subtree t of t_E, $\mathrm{pos}(t)$ denotes the set of positions occurring in t. Observe that if the root of t is the expression F (we write $\mathrm{root}(t) = F$ for short) then $\mathrm{pos}(t) \subseteq \mathrm{pos}(F)$. The inclusion may be strict since t need not be the full subtree t_F of t_E below F. As a measure for t we use the cardinality of $\mathrm{pos}(t)$; we set $|t| = |\mathrm{pos}(t)|$.

Our approach grounds on an analysis of the rules (8)–(11) in Section 3, which define the follow set of a position x. They can be read as follows. To obtain $\mathrm{follow}(E, x)$, climb up in the tree t_E from x to the root (in a bottom-up fashion) and on certain occasions, when $x \in \mathrm{last}(F)$ for some subexpression F, add $\mathrm{first}(F)$ or $\mathrm{first}(G)$ for a related subexpression G, to the follow set of x.

In order to be able to formalize this observation, we define a function "next" between subexpressions of E as

$$\mathrm{next}(F) = \begin{cases} F & \text{if } F \text{ is a son of } F^* \text{ in } t_E, \\ G & \text{if } F \text{ is a son of } FG \text{ in } t_E, \\ \bullet & \text{otherwise.} \end{cases}$$

We set $\mathrm{first}(\bullet) = \varnothing$, and obtain (a) of the following lemma as a reformulation of (8)–(11). The other parts of the lemma result rather directly from the fact that the first and last sets are obtained by computing unions bottom up.

LEMMA 5.1. *Let E be any regular expression and $x \in \mathrm{pos}(E)$. Assume a fixed tree representation t_E of E, and let F, G be any expressions occurring as nodes in t_E. Then the following holds*:

(a) $\text{follow}(F, x) = \bigcup_{H \succ F,\, x \in \text{last}(H)} \text{first}(\text{next}(H))$.

(b) *If* $F \prec G$, *then* $\text{first}(\text{next}(G)) \subseteq \text{pos}(F)$.

(c) *If* $F \preccurlyeq G$, *then* $\text{last}(F) \cap \text{pos}(G) = \text{last}(G)$ *or* $\text{last}(F) \cap \text{pos}(G) = \varnothing$, and the same holds for first in place of last.

Proof. (a) We proceed by structural induction on F.

For a leaf a with a position x, the union on the right-hand side is empty, which coincides with $\text{follow}(a, x) = \varnothing$, according to (8).

In the induction step, we must distinguish the different types of internal nodes.

If $F = F_1 + F_2$, and F_1, F_2 are the sons of F in the chosen decomposition, the same argument applies for both subcases, $x \in \text{pos}(F_i)$, $i \in \{1, 2\}$. Here we have $\text{follow}(F, x) = \text{follow}(F_i, x)$, according to (9), and the union over all $H \succ F$ where $x \in \text{last}(H)$ is the same as over all $H \succ F_i$, except that potentially $H = F_i$ is added. However, by the above definition, $\text{next}(F_i) = \bullet$, and thus the new contribution is empty. Hence our claim transfers directly from the induction hypothesis.

Next, we look at case $F = F_1^*$. We start from the induction hypothesis that our claim holds for F_1 and observe that going from F_1 to F, the right-hand side of the claimed equation is augmented at most by $\text{first}(\text{next}(F_1)) = \text{first}(F_1)$. If $x \notin \text{last}(F_1)$, this set is not added, and in this case we have $\text{follow}(F, x) = \text{follow}(F_1)$ by (11). If $x \in \text{last}(F_1)$, $\text{first}(F_1)$ is added on the right-hand side, and by (11), $\text{follow}(F, x) = \text{follow}(F_1, x) \cup \text{first}(F_1)$.

When $F = F_1 \cdot F_2$, we have two different subcases. The case $x \in \text{pos}(F_2)$ is dealt with just as when $F = F_1 + F_2$, and for $x \in \text{pos}(F_1)$, we copy the argument from case $F = F_1^*$, using $\text{first}(\text{next}(F_1)) = \text{first}(F_2)$ and (10). This proves (a).

Before proceeding to the proofs of the remaining claims, (b) and (c), recall that the position sets of two expressions are disjoint unless one expression is part of the other in the chosen decomposition.

(b) It is obvious that this claim holds for all $F \preccurlyeq G$ if it holds for F being the father of G. Assuming that F is the father of G, we observe that if $\text{next}(G) \neq \bullet$, then $\text{next}(G)$ is either G itself or another son of F. In both cases we have $\text{first}(\text{next}(G)) \subseteq \text{pos}(\text{next}(G)) \subseteq \text{pos}(F)$. If $\text{next}(G) = \bullet$, the claim holds trivially.

(c) Here, we proceed by induction on F ranging over the ancestors of G, starting with $F = G$. In that case, the claim holds trivially.

Assume the claim holds for $F' \preccurlyeq G$ and let F be the father of F'. According to its definition in (4), (6), and (7), $\text{last}(F)$ contains either exactly $\text{last}(F')$ or nothing from $\text{pos}(F')$.

Consequently, if $\text{last}(F') \cap \text{pos}(G) = \varnothing$, we have also $\text{last}(F) \cap \text{pos}(G) = \varnothing$.

Now let $\text{last}(F') \cap \text{pos}(G) = \text{last}(G)$. Case $\text{last}(F) \cap \text{pos}(F') = \text{last}(F')$ implies $\text{last}(F) \cap \text{pos}(G) = \text{last}(G)$, and $\text{last}(F) \cap \text{pos}(F') = \varnothing$ gives $\text{last}(F) \cap \text{pos}(G) = \varnothing$. ∎

From the proof of (c) we also learn the following, which we note for later use.

Remark. For every node G in the tree representation t_E of E, there is some node H, $E \preccurlyeq H \preccurlyeq G$, such that

(a) $\text{last}(F) \cap \text{pos}(G) = \text{last}(G)$ for all F such that $H \preccurlyeq F \preccurlyeq G$, and

(b) $\text{last}(F) \cap \text{pos}(G) = \varnothing$ for all F such that $E \preccurlyeq F \prec H$.

With Lemma 5.1(a) we now have a description of follow(F, x) that can be extended to arbitrary subtrees of t_E. For every subtree t of t_E, we set

$$\text{follow}(t, x) = \text{pos}(t) \cap \bigcup_{\substack{H \succ \text{root}(t) \\ x \in \text{last}(H)}} \text{first}(\text{next}(H)).$$

Lemma 5.1(a) guarantees that this definition is consistent with that from the previous section in the following sense: follow(t_F, x) = follow(F, x) for expressions F and $x \in \text{pos}(F)$.

6. BASIC TRANSLATION

We describe a first way how, for the given regular expression E, one computes a system of CFS that yields a small CFS automaton. The main idea is to recursively compute for certain subtrees t of t_E (starting with $t = t_E$) sets $\mathscr{C}(t) \subseteq \mathscr{P}(\text{pos}(t))$ and decompositions dec(x, t) that satisfy for some $x \in \text{pos}(t)$

$$\text{dec}(x, t) \subseteq \mathscr{C}(t), \tag{12}$$

and

$$\text{follow}(t, x) = \bigcup_{C \in \text{dec}(x, t)} C. \tag{13}$$

We shall say that a decomposition dec is *appropriate* for t and $x \in \text{pos}(t)$ if (13) holds.

Applied to the entire tree, (13) gives by Lemma 5.1(a)

$$\text{follow}(E, x) = \text{follow}(t_E, x) = \bigcup_{C \in \text{dec}(x, t_E)} C \qquad \text{for} \quad x \in \text{pos}(E).$$

Hence we may set

$$\mathscr{C} = \{\text{first}(E)\} \cup \mathscr{C}(t_E) \tag{14}$$

and

$$\text{dec}(x) = \text{dec}(x, t_E) \qquad \text{for} \quad x \in \text{pos}(E), \tag{15}$$

in order to obtain a system of CFS for E.

When x is the only position in t, it is very easy to find values for dec that make it appropriate for a tree t and a position x (although, as we will see later, one can do better than what is described in the following):

$$\text{dec}(x, t) = \begin{cases} \{\{x\}\} & \text{if there exists a node } F \text{ in } t \text{ other than the} \\ & \text{root such that } x \in \text{last}(F) \cap \text{first}(\text{next}(F)), \\ \{\varnothing\} & \text{otherwise.} \end{cases} \tag{16}$$

Next, consider a tree t with $|t| > 1$. Assume t_1 is the subtree of t below some node F_1 and t_2 is the rest of t after removing t_1. Let x be a position in t. We will show later that if $x \in \mathrm{pos}(t_1)$ and dec is appropriate for t_1 and x, then we can set

$$\mathrm{dec}(x, t) = \begin{cases} \mathrm{dec}(x, t_1) \cup \{C_1\} & \text{if} \quad x \in \mathrm{last}(F_1), \\ \mathrm{dec}(x, t_1) & \text{otherwise,} \end{cases} \tag{17}$$

where

$$C_1 = \mathrm{pos}(t) \cap \bigcup_{\substack{F \prec G \leqslant F_1 \\ \mathrm{last}(G) \cap \mathrm{pos}(F_1) = \mathrm{last}(F_1)}} \mathrm{first}(\mathrm{next}(G)), \tag{18}$$

so as to make dec appropriate for t and x. Similarly, we will see that if $x \in \mathrm{pos}(t_2)$ and dec is appropriate for t_2 and x, then we can set

$$\mathrm{dec}(x, t) = \begin{cases} \mathrm{dec}(x, t_2) \cup \{C_2\} & \text{if first } (F_1) \subseteq \mathrm{follow}(E, x), \\ \mathrm{dec}(x, t_2) & \text{otherwise,} \end{cases} \tag{19}$$

where

$$C_2 = \mathrm{pos}(t) \cap \mathrm{first}(F_1), \tag{20}$$

so as to make dec appropriate for t and x. Observe that C_1 and C_2 are independent of the particular position x.

Now that we know how to find appropriate decompositions for small trees and how to combine appropriate decompositions for trees to appropriate decompositions for larger trees it is easy to come up with a first algorithm that computes a decomposition for t_E, using a balanced strategy. It divides repeatedly trees into two parts, each of which has size at least 1/3 of the size of the divided tree.

ALGORITHM 1 (t subtree of t_E).

1. If $|t| = 1$, compute the decomposition according to (16).
2. If $|t| > 1$, carry out the following steps.
3. Starting from the root of t, search downward for some node F_1 such that $\frac{|t|}{3} \leqslant |t_1| \leqslant \frac{2|t|}{3}$ where t_1 is the subtree of t below F_1. (Such a node F_1 does always exist in a binary tree.) Let t_2 be the rest of t after removing t_1.
4. Recursively apply Algorithm 1 to t_1 and t_2.
5. For every $x \in \mathrm{pos}(t)$ determine $\mathrm{dec}(x, t)$ according to (17)–(20).

This algorithm produces an automaton with a small number of transitions:

LEMMA 6.1. *Let* $\mathscr{C} = \{\mathrm{first}(E)\} \cup \mathscr{C}(t_E)$ *and assume* $\mathrm{dec}(x) = \mathrm{dec}(x, t_E)$ *is computed according to Algorithm* 1.

Then

(a) $|\mathscr{C}| \leqslant 3n - 1$,

(b) $\sum_{C \in \mathscr{C}} |C| \leqslant \frac{3}{\log 3/2}\, n \log n + n + 1$, *and*

(c) $|\mathrm{dec}(x)| \leqslant \frac{1}{\log 3/2} \log n + 1$ *for all* $x \in \mathrm{pos}(E)$.

We do not prove this here, as we will soon see an algorithm with smaller constants. If we would apply the construction of the CFS automaton (Definition 4.1) to the CFS system obtained so far, we would get by Lemma 6.1 an automaton having at most $6n-2$ states and about

$$\frac{6}{(\log_2 3/2)^2}\, n(\log n)^2$$

transitions.

By refining the construction in the following, we will obtain an automaton having at most $2n-1$ states (Lemma 9.2) and

$$\frac{4}{(\log_2 3/2)^2}\, n(\log_2 n)^2$$

transitions (Lemma 7.1).

7. REFINED TRANSLATION

To obtain better bounds on the size of the resulting automata, we refine our strategy for computing decompositions. We add one more parameter: a set P of positions of the given tree t_E.

The following algorithm computes for a given subtree t small decompositions for the elements of $\mathrm{pos}(t) \cap P$. Given a tree t, $|t|_P$ denotes $|\mathrm{pos}(T) \cap P|$.

ALGORITHM 2 (t subtree of t_E with root F, $P \subseteq \mathrm{pos}(E)$).

1. If $|t|_P = 1$, compute the decomposition according to (16).
2. If $|t|_P > 1$, carry out the following steps.
3. Starting from the root of t, search downward for some node F_1 such that $\frac{|t|_P}{3} \leqslant |t_1|_P \leqslant \frac{2\,|t|_P}{3}$ where t_1 is the subtree of t below F_1.
4. Let t_2 be the rest of t after removing t_1.
5. Recursively apply Algorithm 2 to t_1 and t_2.
6. For every $x \in \mathrm{pos}(t) \cap P$ determine $\mathrm{dec}(x, t)$ according to (17)–(20).
7. If $|t|_P \in \{2, 3\}$, then do the following for every $i \in \{1, 2\}$.
   If $|t_i|_P = 1$ and $\mathrm{dec}(x, t) = \{C, C'\}$ for some sets C and C' then let $\mathrm{dec}(x, t) = \{C \cup C'\}$.
8. If $|\mathrm{dec}(x, t)| \geqslant 2$, then let $\mathrm{dec}(x, t) = \mathrm{dec}(x, t) \setminus \{\varnothing\}$.

Step 7 is justified as follows. If $|t_i|_P = 1$, the set C_i is to be used only (if at all) in the decomposition for the single $x \in \mathrm{pos}(t_i) \cap P$, see (17) and (19). In a recursive call on t_i, another set C_i' is computed to be used only (if at all) in the decomposition for the same $x \in \mathrm{pos}(t_i) \cap P$, see (16). We may very well replace C_i and C_i' by their union $C_i \cup C_i'$, one of C_i, C_i', or the empty set (depending on which of the two sets belong to $\mathrm{dec}(x, t)$ according to (17), (19), (16)).

Step 8 is a trivial optimization. Unless a decomposition consists of the empty set only, the empty set can be removed from any decomposition.

In order to obtain a full decomposition for E, we call this algorithm with two sets, P_0 and P_1, whose union is $\mathrm{pos}(E)$, so as to obtain a decomposition for every

position of E. Specifically, we choose $P_0 = \mathrm{pos}(E) \backslash \mathrm{last}(E)$ and $P_1 = \mathrm{last}(E)$. Say dec_0 and dec_1 are the partial decompositions one obtains and $\mathscr{C}_0$ and $\mathscr{C}_1$ the corresponding families of CFS. We define a joint decomposition dec by setting

$$\mathrm{dec}(x) = \begin{cases} \mathrm{dec}_0(x), & x \in \mathrm{pos}(E) \backslash \mathrm{last}(E) \\ \mathrm{dec}_1(x), & x \in \mathrm{last}(E) \end{cases} \tag{21}$$

for every position $x \in \mathrm{pos}(E)$, and we will prove in Section 9 that this leads to an automaton satisfying the bounds on the number of states and transitions stated in the lemma below.

The idea is to make a distinction between final and nonfinal states right from the beginning (remember the $\mathrm{last}(C)$ is the set of final states in the position automaton), rather than producing a final and a nonfinal copy of each state at the end. Choosing $\mathrm{last}(E)$ and $\mathrm{pos}(E) \backslash \mathrm{last}(E)$ makes sure, as we will see in Section 9, that all reachable states in the CFS automaton based on dec belong to $\{(\mathrm{first}(E), f_\varepsilon)\} \cup (\mathscr{C}_0 \times \{0\}) \cup (\mathscr{C}_1 \times \{1\})$. There, we will give the proof of the following lemma.

LEMMA 7.1. *Let* $n = |\mathrm{pos}(E)| \geqslant 2$, *and let* dec *be defined according to* (21). *The CFS automaton based on* dec *has at most* $(4/(\log_2 3/2)^2)\, n(\log_2 n)^2$ *transitions, when restricted to reachable states.*

Let us remark that Volker Diekert pointed out to us that we could have used

$$C_i = \mathrm{pos}(t_{3-i}) \cap \mathrm{follow}(E, x) \tag{22}$$

instead of (17) and (19). (One can show that C_i is the same set for *all* positions with $\mathrm{pos}(t_{3-i}) \cap \mathrm{follow}(E, x) \neq \varnothing$.) The method we suggest allows a faster computation of the entire CFS automaton, as we will explain in Section 10.

We conclude this section by demonstrating the algorithm briefly. We show how it produces the small NFA for E_5 depicted in Fig. 1. We started from the expression $(a_1 + \varepsilon) \cdot ((a_2 + \varepsilon) \cdot ((a_3 + \varepsilon) \cdot ((a_4 + \varepsilon) \cdot (a_5 + \varepsilon))))$. Since $\mathrm{last}(E_5) = \mathrm{pos}(E_5)$, Algorithm 2 must be applied only once, to $t = t_{E_5}$ and $P = \mathrm{pos}(E_5)$. Consequently, the resulting automaton has only final states.

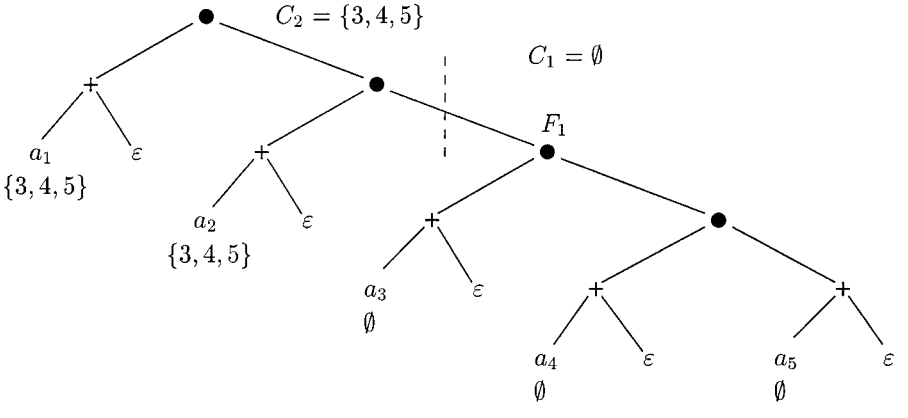

FIG. 2. Example decomposition of a regular expression tree.

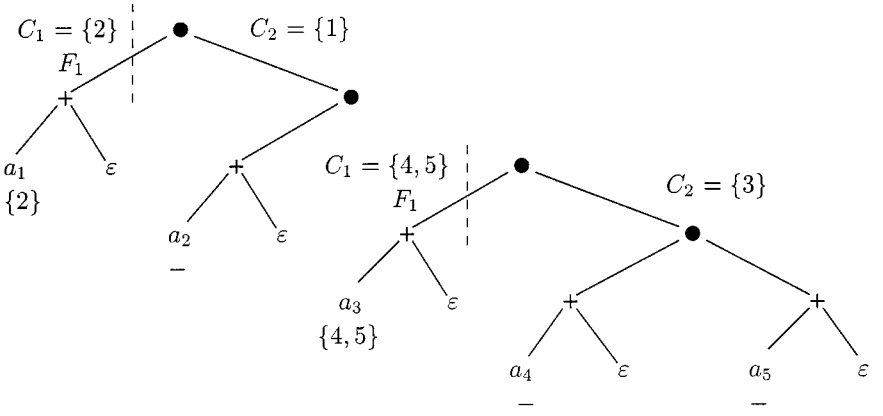

FIG. 3. Example decomposition of a regular expression tree (continued).

In Figs. 2–4, it is shown how the corresponding tree t_{E_5} will be successively divided by our algorithm.

Figure 2 shows the topmost division of t_{E_5}. F_1 is the first node whose subtree size is between 1/3 and 2/3 of the whole tree size. The dashed line marks the resulting division of t_{E_5}, and the sets C_1 and C_2 are calculated according to Eq. (18) and (20). Then, below each position x, one of these sets (or none) is marked, which is added to the respective dec(x), according to rules (17), respectively (19).

Similarly, Fig. 3 shows in parallel for both subtrees how these are divided further. Let us remark that we could have chosen F_1 to be the right instead of the left son of the root in both cases, without changing the overall result.

The division is continued in Fig. 4 where for the trees having size 1 already, rule (16) is applied. We omitted the last step, where for the last two subtrees of size 1 a contribution of $\varnothing$ to the decomposition is obtained. Remember that Step 8 of Algorithm 2 leaves $\varnothing$ in the final decomposition only if it is the only element, as is the case for Position 5.

The resulting decomposition sets dec(x) are shown in Table (23). From this, the automaton of Fig. 1 is obtained immediately by Definition 4.1.

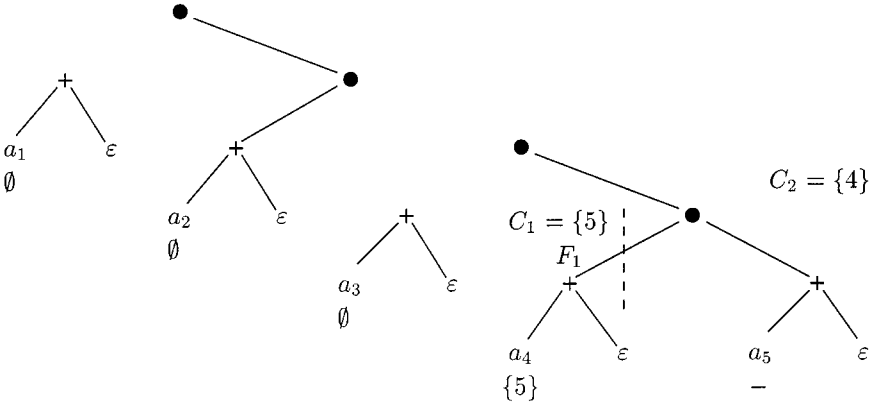

FIG. 4. Example decomposition of a regular expression tree (continued).

position x	dec(x)
1	$\{2\}, \{3, 4, 5\}$
2	$\{3, 4, 5\}$
3	$\{4, 5\}$
4	$\{5\}$
5	$\varnothing$

(23)

8. PROOF OF CORRECTNESS

The correctness of Algorithms 1 and 2 follow directly from the lemma below.

LEMMA 8.1. *Let t be a subtree of t_E, t_1 the subtree of t below some node F_1, and t_2 the rest of t after removing t_1, such that $|t_1|, |t_2| \geqslant 1$.*

(a) *If $x \in \mathrm{pos}(t_1)$ and* dec *is appropriate for t_1 and x, then setting* dec *according to* (17) *and* (18) *makes* dec *appropriate for t and x.*

(b) *If $x \in \mathrm{pos}(t_2)$ and* dec *is appropriate for t_2 and x, then setting* dec *according to* (19) *and* (20) *makes* dec *appropriate for t and x.*

Proof. (a) We start by using the assumption that there is an appropriate decomposition for follow(t_1, x), i.e.,

$$\mathrm{follow}(t_1, x) \underset{\text{(def.)}}{=} \mathrm{pos}(t_1) \cap \bigcup_{\substack{H \succ F_1 \\ x \in \mathrm{last}(H)}} \mathrm{first}(\mathrm{next}(H)) = \bigcup_{C \in \mathrm{dec}(x,\, t_1)} C. \tag{24}$$

Lemma 5.1(b) implies that the intersection may be taken w.r.t. pos(t) instead of pos(t_1):

$$\mathrm{pos}(t) \cap \bigcup_{\substack{H \succ F_1 \\ x \in \mathrm{last}(H)}} \mathrm{first}(\mathrm{next}(H)) = \bigcup_{C \in \mathrm{dec}(x,\, t_1)} C. \tag{25}$$

By definition, follow(t, x) is the union of the left-hand side of (25) with

$$\mathrm{pos}(t) \cap \bigcup_{\substack{F \prec G \preccurlyeq F_1 \\ x \in \mathrm{last}(G)}} \mathrm{first}(\mathrm{next}(G)). \tag{26}$$

It remains to show that (26) equals either C_1 or $\varnothing$, depending on whether $x \in \mathrm{last}(F_1)$.

In case $x \in \mathrm{last}(F_1)$, by (18) it is sufficient to show that the condition $x \in \mathrm{last}(G)$ is equivalent to $\mathrm{last}(G) \cap \mathrm{pos}(F_1) = \mathrm{last}(F_1)$ for $x \in \mathrm{last}(F_1)$. This equivalence is provided by Lemma 5.1(c).

If $x \in \mathrm{pos}(t_1) \backslash \mathrm{last}(F_1)$, we want to show that the union in (26) is an empty one. This in turn is again a consequence of Lemma 5.1(c), which states that if $x \notin \mathrm{last}(F_1)$, then $x \notin \mathrm{last}(G)$ for all $G \prec F_1$.

(b) Looking at $\text{follow}(t, x) = \text{pos}(t) \cap \bigcup_{H \succ F,\, x \in \text{last}(H)} \text{first}(\text{next}(H))$, we first observe that the path from the root to any leaf of t_2 is unchanged compared to t. This means that the large union is taken over the same nodes. The only difference between $\text{follow}(t, x)$ and $\text{follow}(t_2, x)$ may thus result from taking different intersections with $\text{pos}(t)$ and $\text{pos}(t_2) = \text{pos}(t) \setminus \text{pos}(t_1)$, respectively.

Next, we convince ourselves that it suffices to examine those $H \succ F$, $x \in \text{last}(H)$ in the big union where $\text{next}(H) \preccurlyeq F_1$. In any other case, i.e., if $\text{next}(H)$ and F_1 are incomparable in the tree, the sets $\text{pos}(\text{next}(H))$ and $\text{pos}(F_1)$ are by definition incomparable, too. This implies $\text{first}(\text{next}(H)) \cap \text{pos}(t_1) = \varnothing$, and hence $\text{first}(\text{next}(H)) \cap \text{pos}(t) = \text{first}(\text{next}(H)) \cap \text{pos}(t_2)$.

For $\text{next}(H) \preccurlyeq F_1$, we know by Lemma 5.1(c) that $\text{pos}(t_1) \cap \text{first}(\text{next}(H))$ is either $\text{pos}(t_1) \cap \text{first}(F_1)$ or $\varnothing$ (recall that $\text{pos}(t_1) \subseteq \text{pos}(F_1)$).

If case $\text{pos}(t_1) \cap \text{first}(\text{next}(H)) = \text{pos}(t_1) \cap \text{first}(F_1)$ occurs for some $H \succ F$, $x \in \text{last}(H)$, then

$$\begin{aligned}
\text{follow}(t, x) &= \text{pos}(t) \cap \bigcup_{\substack{H \succ F \\ x \in \text{last}(H)}} \text{first}(\text{next}(H)) \\
&= \text{pos}(t_2) \cap \bigcup_{\substack{H \succ F \\ x \in \text{last}(H)}} \text{first}(\text{next}(H)) \cup \text{pos}(t_1) \cap \text{first}(F_1) \\
&= \bigcup_{C \in \text{dec}(x,\, t_2)} C \cup C_2 \\
&= \text{follow}(t_2, x) \cup C_2.
\end{aligned}$$

If case $\text{pos}(t_1) \cap \text{first}(\text{next}(H)) = \varnothing$ applies to all $H \succ F$, $x \in \text{last}(H)$, this means that $\text{follow}(t, x) \cap \text{pos}(t_1) = \varnothing$, hence $\text{follow}(t, x) = \text{follow}(t_2, x)$. ∎

From the last steps of the proof, we get the following observation.

Remark. The condition $\text{first}(F_1) \subseteq \text{follow}(E, x)$ in (19) is equivalent to the existence of some node H such that $x \in \text{last}(H)$, $\text{next}(H) \preccurlyeq F_1$, and $\text{pos}(t_1) \cap \text{first}(\text{next}(H)) = \text{pos}(t_1) \cap \text{first}(F_1)$.

9. DESCRIPTIVE COMPLEXITY

We begin with more terminology. The *height* of a decomposition, denoted $h(\text{dec})$, is the maximal number of subtrees used in the calculation of dec that contain the same position. Inductively, we define

$$h(\text{dec}, t) = \begin{cases} 0 & \text{if } |t| = 1, \\ \max(h(\text{dec}, t_i), h(\text{dec}, t_2)) + 1 & \text{if } t \text{ is divided into } t_1 \text{ and } t_2 \end{cases} \tag{27}$$

$$h(\text{dec}) = h(\text{dec}, t_E).$$

Note that we start with height 0 since the common follow sets calculated for subtrees of size 1 are hidden in the final decomposition after merging (Algorithm 2, Step 7). Thus, the above definition is made in such a way that

$$\max_{x \in \mathrm{pos}(t) \cap P} |\mathrm{dec}(x, t)| \leqslant h(\mathrm{dec}, t), \qquad \text{if} \quad |\mathrm{pos}(t) \cap P| > 1. \tag{28}$$

Now, we are in position to prove the central complexity lemma.

LEMMA 9.1. *Let E be any regular expression, t_E a tree decomposition for E, and $P \subseteq \mathrm{pos}(E)$ such that $|P| \geqslant 2$. Assume* dec *is computed by Algorithm* 2 *applied to t and P, and define $\mathscr{C}$ to be $\bigcup_{x \in P} \mathrm{dec}(x)$.*

Then

(a) $|\mathscr{C}| \leqslant 2\,|P| - 2$,

(b) $h(\mathrm{dec}) \leqslant \log_{3/2} |P|$, *and*

(c) $\sum_{C \in \mathscr{C}} |C| \leqslant 2h(\mathrm{dec})\,|\mathrm{pos}(E)|$.

Proof. We will show the corresponding claims for all subtrees t used in calculating the decomposition. That is, we must substitute $\mathscr{C}$ by $\mathscr{C}(t)$, P by $\mathrm{pos}(t) \cap P$, $h(\mathrm{dec})$ by $h(\mathrm{dec}, t)$, and $\mathrm{pos}(E)$ by $\mathrm{pos}(t)$ in the above claims. We proceed by induction on $|t| = |\mathrm{pos}(t) \cap P|$ where $|t| \geqslant 2$.

In case $|t| = 2$, t is divided by Algorithm 2 into two subtrees t_1, t_2, each containing one element (x_1, resp. x_2) of P. The corresponding sets C_1, $C_2 \subseteq \mathrm{pos}(t)$ are merged with the respective sets $C_1' \subseteq \mathrm{pos}(t_1)$, and $C_2' \subseteq \mathrm{pos}(t_2)$, calculated on the respective subtrees, into C_1'', and C_2'', respectively. Thus, we obtain for each $x_i \in \mathrm{pos}(t) \cap P$ a single set $C_i'' \subseteq \mathrm{pos}(t)$ in $\mathrm{dec}(x, t)$ (after merging). Consequently,

$$|\mathscr{C}(t)| \leqslant 2 = 2\,|\mathrm{pos}(t) \cap P| - 2,$$

$$h(\mathrm{dec}, t) = 1 \leqslant \log_{3/2} 2 = \log_{3/2} |\mathrm{pos}(t) \cap P|,$$

$$\sum_{C \in \mathscr{C}(t)} |C| \leqslant 2\,|\mathrm{pos}(t)| = 2h(\mathrm{dec}, t)\,|\mathrm{pos}(t)|.$$

In the case $|t| = 3$, the subtrees are of size 1 and 2 respectively, say $|t_i| = 2$ and $|t_{3-i}| = 1$, $i \in \{1, 2\}$.

For t_{3-i}, we obtain as before a single set $C_{3-i} \subseteq \mathrm{pos}(t)$, after merging. For t_i there can belong to a decomposition of $x \in \mathrm{pos}(t_i) \cap P$ at most C_i and the single sets from $\mathrm{dec}(x, t_i)$, see the previous case. This gives

$$|\mathscr{C}(t)| \leqslant 4 = 2\,|\mathrm{pos}(t) \cap P| - 2,$$

$$h(\mathrm{dec}, t) = 2 < \log_{3/2} 3 = \log_{3/2} |\mathrm{pos}(t) \cap P|,$$

$$\sum_{C \in \mathscr{C}(t)} |C| \leqslant 2\,|\mathrm{pos}(t)| + 2\,|\mathrm{pos}(t_i)| \leqslant 2h(\mathrm{dec}, t)\,|\mathrm{pos}(t)|.$$

If $|t| > 3$, both subtrees, t_1 and t_2 are covered by the induction hypothesis, since $1 < \frac{1}{3}|t| \leqslant |t_1|$, $|t_2| \leqslant \frac{2}{3}|t|$. The last inequality also implies that for $i \in \{1, 2\}$ $\log_{3/2} |\text{pos}(t_i) \cap P| \leqslant \log_{3/2} |\text{pos}(t) \cap P| - 1$. Hence

$$\begin{aligned} |\mathscr{C}(t)| &\leqslant 2\,|\text{pos}(t_1) \cap P| - 2 + 2\,|\text{pos}(t_2) \cap P| - 2 + 2 \\ &= 2\,|\text{pos}(t) \cap P| - 2, \\ h(\text{dec}, t) &= \max_{i=1,2} h(\text{dec}, t) + 1 \leqslant \max_{i=1,2} \log_{3/2} |\text{pos}(t_i) \cap P| + 1 \\ &\leqslant \log_{3/2} |\text{pos}(t) \cap P|, \end{aligned}$$

and

$$\begin{aligned} \sum_{C \in \mathscr{C}(t)} |C| &\leqslant 2\,|\text{pos}(t)| + \sum_{C \in \mathscr{C}(t_1)} |C| + \sum_{C \in \mathscr{C}(t_2)} |C| \\ &\leqslant 2\,|\text{pos}(t)| + 2h(\text{dec}, t_1)\,|\text{pos}(t_1)| + 2h(\text{dec}, t_2)\,|\text{pos}(t_2)| \\ &\leqslant 2\,|\text{pos}(t)| + 2 \max_{i=1,2} h(\text{dec}, t_1)\,|\text{pos}(t)| = 2h(\text{dec}, t)\,|\text{pos}(t)|. \end{aligned}$$

For $t = t_E$, this gives the claim of the lemma. ∎

In the following, let dec, $\mathscr{C}_0$ and $\mathscr{C}_1$ be as described after Algorithm 2.

LEMMA 9.2. *The CFS automaton based on* dec *has at most* $2 \cdot \text{size}(E) - 1$ *reachable states.*

Proof. The state set of the CFS automaton is defined as $Q = \mathscr{C} \times \{0, 1\}$ where $\mathscr{C} = \{\text{first}(E)\} \cup \bigcup_{x \in \text{pos}(E)} \text{dec}(x)$. Since first($E$) is used for the initial state only in one of the two possible cases, (first(E), 0) and (first(E), 1), it suffices to show that from $(\mathscr{C}_0 \cup \mathscr{C}_1) \times \{0, 1\}$, there are at most $2\,|\text{pos}(E)| - 2$ reachable states.

For this it suffices to show that $|\mathscr{C}_0| + |\mathscr{C}_1| \leqslant 2\,|\text{pos}(E)| - 2$, and that any reachable state is in $\mathscr{C}_0 \times \{0\} \cup \mathscr{C}_1 \times \{1\}$.

The latter claim is guaranteed by the definition of the transition relation of the CFS automaton. Transitions into $(C, 1)$ occur only if $C \in \text{dec}(x)$ for some $x \in \text{last}(E)$, and transitions into $(C, 0)$ occur only if $C \in \text{dec}(y)$ for some $y \in \text{pos}(E) \backslash \text{last}(E)$.

The claim $|\mathscr{C}_0| + |\mathscr{C}_1| \leqslant 2\,|\text{pos}(E)| - 2$ follows rather directly from Lemma 9.1(a). If $\text{pos}(E) = \text{last}(E)$, then $|\text{last}(E)| \geqslant 2$, and hence

$$|\mathscr{C}_0| + |\mathscr{C}_1| = |\mathscr{C}_1| \underset{\text{Lemma 9.1(a)}}{\leqslant} 2\,|\text{pos}(E)| - 2.$$

Otherwise each of last(E) and pos(E)\last(E) contain at least one element. Remember that for $|P| = 1$ ($P = \text{last}(E)$ or $P = \text{pos}(E) \backslash \text{last}(E)$) the corresponding set $\mathscr{C}_i$, $i \in \{1, 2\}$, contains exactly one element. Therefore, we have in any case (applying Lemma 9.1(a) for $|P| \geqslant 2$) $\mathscr{C}_i \leqslant 2\,|P| - 1$. Consequently,

$$|\mathscr{C}_0| + |\mathscr{C}_1| \leqslant 2\,|\text{last}(E)| - 1 + 2\,|\text{pos}(E) \backslash \text{last}(E)| - 1 = 2\,|\text{pos}(E)| - 2. \quad ∎$$

Proof of Lemma 7.1. We look at the outgoing transitions of a reachable state $(C, f) \in Q$ of the CFS automaton. According to its definition, the CFS automaton has one transition $((C, f), \langle E \rangle_x, (C', f_x))$ for each $x \in C$, and each $C' \in \mathrm{dec}(x)$.

That gives the number of edges, when restricted to reachable states, as

$$|\Delta| = \sum_{(C, f) \text{ reachable}} \sum_{x \in C} |\mathrm{dec}(x)|.$$

This can be bounded by

$$|\Delta| \leqslant \sum_{(C, f) \text{ reachable}} |C| \max_{x \in \mathrm{pos}(E)} |\mathrm{dec}(x)|.$$

We have determined the set of reachable states as $\{(\mathrm{first}(E), f_\varepsilon)\} \cup \mathscr{C}_1 \times \{1\} \cup \mathscr{C}_0 \times \{0\}$. Using Lemma 9.1(c), we get

$$\begin{aligned} |\Delta| &\leqslant \left(|\mathrm{first}(E)| + \sum_{C \in \mathscr{C}_1} |C| + \sum_{C \in \mathscr{C}_0} |C| \right) \max_{x \in \mathrm{pos}(E)} |\mathrm{dec}(x)| \\ &\leqslant (|\mathrm{first}(E)| + 2h(\mathrm{dec}_1)\,|\mathrm{pos}(E)| + 2h(\mathrm{dec}_0)\,|\mathrm{pos}(E)|) \max_{x \in \mathrm{pos}(E)} |\mathrm{dec}(x)|. \end{aligned} \tag{29}$$

By Lemma 9.1(b), we have $h(\mathrm{dec}_1) \leqslant \log_{3/2} |\mathrm{last}(E)|$ and $h(\mathrm{dec}_0) \leqslant \log_{3/2} |\mathrm{pos}(E) \backslash \mathrm{last}(E)|$. Consequently, $h(\mathrm{dec}_1) + h(\mathrm{dec}_0)$ can be bound by

$$\begin{aligned} h(\mathrm{dec}_1) + h(\mathrm{dec}_0) &\leqslant 2 \log_{3/2}(|\mathrm{pos}(E)|/2) \\ &= 2 \log_{3/2} |\mathrm{pos}(E)| - 2 \log_{3/2} 2 \\ &\leqslant 2 \log_{3/2} |\mathrm{pos}(E)| - 3. \end{aligned} \tag{30}$$

Lemma 9.1(b) is also used to estimate $\max_{x \in \mathrm{pos}(E)} |\mathrm{dec}(x)|$:

$$\max_{x \in \mathrm{pos}(E)} |\mathrm{dec}(x)| \underset{(28)}{\leqslant} \max_{i \in \{0, 1\}} h(\mathrm{dec}_i) \underset{\text{La.9.1(b)}}{\leqslant} \log_{3/2} |\mathrm{pos}(E)| \tag{31}$$

We finish the proof by inserting (30) and (31) into (29)

$$\begin{aligned} |\Delta| &\leqslant (|\mathrm{pos}(E)| + 2(2 \log_{3/2} |\mathrm{pos}(E)| - 3)\,|\mathrm{pos}(E)|) \log_{3/2} |\mathrm{pos}(E)| \\ &\leqslant 4\,|\mathrm{pos}(E)|\,(\log_{3/2} |\mathrm{pos}(E)|)^2 \\ &= \frac{4}{(\log_2 3/2)^2}\, n(\log_2 n)^2. \quad \blacksquare \end{aligned}$$

10. EFFICIENT IMPLEMENTATION

We calculate the time needed for constructing an NFA from a regular expression of size $n \geqslant 2$. As described in the previous sections, our construction mainly consists

of two applications of Algorithm 2. Thus, if we can show that either application can be performed in time $\mathcal{O}(n^2 \log n)$, then $\mathcal{O}(n^2 \log n)$ is also a bound for the time used by the whole construction, for the following reason. The only other additional effort consists in calculating the first and last sets and building the automaton out of the decomposition at the end. According to (1)–(7), first and last sets can be obtained by $\mathcal{O}(n)$ unions of sets of size at most n, and the automaton is obtained by a straightforward realization of Definition 4.1 in time proportional to its size, that is, $\mathcal{O}(n(\log n)^2)$.

In the following, we calculate the time needed by Algorithm 2 when applied on t_E and some $P \subseteq \text{pos}(E)$. This is measured in terms of T_u and T_i, where T_u is the time needed to compute the union of two sets of positions and T_i is the time needed to compute the intersection of two sets of positions. Each of these two operations can be performed in time at most $\mathcal{O}(n)$, but potentially there may be more efficient implementations for our particular application.

First, we want to check the time this algorithm takes at a single call, i.e., without counting the recursive calls of itself. Remember that the algorithm is recursively called $\mathcal{O}(n)$ times on the $\mathcal{O}(n)$ subtrees of t_E used in the decomposition.

Step 4 can be performed in constant time. The same holds for realizing (17) in Step 6. Calculating C_2 according to (20) in Step 6 takes time T_i, and Step 7 takes at most T_u. Summed up over all recursive calls, these efforts can be bound by $\mathcal{O}(n(T_i + T_u))$.

Step 3 depends on the length of a path from the root of t to the root F_1 of t_1. Such a path can have linear size, but we know that every node of t_E occurs in at most $\mathcal{O}(\log n)$ subtrees. Thus, the summed up size over all paths used in all recursive calls is in $\mathcal{O}(n \log n)$, and all executions of Step 3 need time $\mathcal{O}(n \log n)$ together.

Similarly, the computation of all sets C_1 according to (18) in Step 6 takes $\mathcal{O}(n \log n T_u)$, when we sum up over all recursive calls.

Finally, for performing test (16) in Step 1, and (19) in Step 6, we make use of Remark 5 and Remark 8. Let us first consider test (19) in Step 6. Remark 8 allows reducing the test in (19) to the existence of some node H such that $x \in \text{last}(H)$, $\text{next}(H) \preccurlyeq F_1$, and $\text{pos}(t_1) \cap \text{first}(\text{next}(H)) = \text{pos}(t_1) \cap \text{first}(F_1)$. According to Remark 5, the last condition on $\text{next}(H)$ holds exactly for all ancestors of F_1 up to some H_1. These nodes H_1 can be easily obtained during the calculation of the first and last sets. Now for realizing (19), we need again only to check the path from the root of t to F_1 for appropriate nodes H (resp. $\text{next}(H)$—remember that $\text{next}(H) \neq \bullet$ is either H itself or its brother). For each such H, we add C_2 to $\text{dec}(x, t)$ for all $x \in \text{pos}(t_2) \cap \text{last}(H)$. Overall, the mentioned test needs only time $\mathcal{O}(n \log n T_i)$, summed up over all recursive called. Test (16) in Step 1 needs similarly only a check along the path from the root of t to the leaf with its single position, which can be bound in the same way.

Thus the running time of the algorithm, summing over all recursive calls, is bounded by $\mathcal{O}((T_u + T_i)(n \log n)) \subseteq \mathcal{O}(n^2 \log n)$.

We finally remark that Hagenah and Muscholl [7] gave an implementation of our original method [9] (essentially the basic translation of Section 6) that runs in time $\mathcal{O}(n(\log n)^2)$. Additionally, they presented a very efficient parallel implementation.

11. LOWER BOUND

In this section, we proof a lower bound for the conversion of regular expressions into NFAs:

THEOREM 11.1. *Let $n > 0$, and define the regular expression E_n by*

$$E_n = (a_1 + \varepsilon)(a_2 + \varepsilon) \cdots (a_n + \varepsilon). \tag{32}$$

Every NFA that recognizes $\mathscr{L}(E_n)$ has at least $n+1$ states and $\frac{n \log n - 3n}{4}$ transitions.

Proof. For every positive integer $n > 0$, write L_n for $\mathscr{L}(E_n)$, and assume $\mathscr{A} = (Q, q_I, \Delta, Q_F)$ is an NFA recognizing L_n. We want to show that $\mathscr{A}$ has at least $n+1$ states and $\frac{n \log n - 3n}{4}$ transitions.

Without loss of generality, assume that each state in Q in reachable and productive; i.e., for each state q there is a path from q_I to q and there exists $q' \in F$ such that there is a path from q to q'.

We partition the set Q into $n+1$ disjoint sets, denoted $Q_1, ..., Q_{n+1}$. For every state q, we first set

$$P_q = \{ j \mid \exists q' \, (q, a_j, q') \in \Delta \}, \tag{33}$$

and then define

$$Q_i = \{ q \mid P_q \neq \varnothing \text{ and } \min P_q = i \} \qquad \text{for} \quad i \in \{1, ..., n\}, \tag{34}$$

$$Q_{n+1} = \{ q \mid P_q = \varnothing \}. \tag{35}$$

Clearly, the sets $Q_1, ..., Q_{n+1}$ form a partition of Q. For convenience in notation, we will use $Q_{\leqslant i}$ and $Q_{\geqslant j}$ as shorthands for $Q_1 \cup Q_2 \cup \cdots \cup Q_i$ and $Q_j \cup Q_{j+1} \cup \cdots \cup Q_{n+1}$, respectively.

Using this notation, we can now state a fundamental property of the transition table of $\mathscr{A}$:

$$\text{for all } (p, a_i, q) \in \Delta \text{ we have } p \in Q_{\leqslant i} \text{ and } q \in Q_{\geqslant i+1}. \tag{36}$$

To see this, assume $(p, a_i, q) \in \Delta$. Then, by definition, $i \in P_p$, hence $p \in Q_{\leqslant i}$. On the other hand, assume for contradiction that q does not belong to $Q_{\geqslant i+1}$. Then there is a transition $(q, a_j, q') \in \Delta$ such that $j \leqslant i$. Since we assume p is reachable and q' is productive, $\mathscr{A}$ accepts a wort that has $a_i a_j$ as a substring—an obvious contradiction, for $j \leqslant i$.

Using (36), it is now easy to show that each Q_i is nonempty, which also implies that $\mathscr{A}$ has at least $n+1$ states. Let us look at a successful run for the word $a_1 a_2 \cdots a_n$, say $q_1, ..., q_{n+1}$ is the sequence of states $\mathscr{A}$ assumes in such a run. First, $q_1 \in Q_1$. Second, for $i \in \{2, ..., n\}$, we get $q_i \in Q_{\leqslant i} \cap Q_{\geqslant i}$ (observe that (q_{i-1}, a_{i-1}, q_i), $(q_i, a_i, q_{i+1}) \in \Delta$), which implies $q_i \in Q_i$. Finally, $q_{n+1} \in Q_{n+1}$, because if there was an edge leaving q_{n+1}, then $\mathscr{A}$ would accept a word that would have

$a_1 a_2 \cdots a_n$ as a strict prefix (remember that we assume all states are productive) but no such word belongs to L_n.

For showing the lower bound on the number of transitions, we classify transitions according to their "length", if (p, a_j, q) is a transition such that $p \in Q_i$ and $q \in Q_k$, we say that $k-i$ is the *length* of (p, a_j, q).

For $k \leqslant \log n$ and $j \in \{1, ..., 2^k\}$, let

$$w_{k,j} = a_j a_{j+2^k} a_{j+2\cdot 2^k} a_{j+3\cdot 2^k} \cdots a_{j+l_{k,j}\cdot 2^k}, \tag{37}$$

where $l_{k,j}$ is maximal such that $j + l_{k,j} \cdot 2^k \leqslant n$. Obviously, each word $w_{k,j}$ belongs to L_n. So we can pick a successful run $\rho_{k,j}$ of $\mathscr{A}$ on each such word. Let $R_{k,j}$ be the set of transitions occurring in $\rho_{k,j}$.

We will show that the following holds for $k \leqslant \log n$ and $j \in \{1, ..., 2^k\}$.

(a) At least one of every two consecutive transitions in $\rho_{k,j}$ is of length greater than 2^{k-1}.

(b) Each transition in $R_{k,j}$ is of length less than 2^{k+1}.

Before we turn to the proof of (a) and (b), we explain why they imply the desired lower bound for the number of transitions in $\mathscr{A}$.

First note that the letter a_i occurs in $w_{k,j}$ if and only if $j = i \bmod 2^k$. This means that $R_{k,j} \cap R_{k,j'} = \varnothing$ for $k \leqslant \log n$ and $j, j' \in \{1, ..., 2^k\}$, provided $j \neq j'$, and it also implies that

$$\sum_{j=1}^{2^k} |w_{k,j}| = n \qquad \text{for} \quad k \leqslant \log n. \tag{38}$$

We conclude from (a) and (b) that $R_{k,j}$ contains at least $(|w_{j,k}| - 1)/2$ transitions of length greater than 2^{k-1} and less than 2^{k+1}. Let $R'_{k,j}$ denote the set of all such transitions. Obviously, $R'_{k,j} \cap R'_{k',j'} = \varnothing$ for $k' \leqslant k-2$ or $k' \geqslant k+2$ and arbitrary $j \in \{1, ..., 2^k\}$, $j' \in \{1, ..., 2^{k'}\}$. In other words, for every two distinct pairs (k, j) and (k', j') from $\{0, ..., \lfloor \log n/2 \rfloor\} \times \{1, ..., 2^{2k}\}$, the sets $R'_{2k,j}$ and $R'_{2k',j'}$ are disjoint sets of transitions.

Let us estimate the total number of elements in all these sets, which is a lower bound for the number of transitions in Δ. For every $k \leqslant \log n/2$, we get, using (38),

$$\sum_{j=1}^{2^{2k}} |R'_{2k,j}| \geqslant \sum_{j=1}^{2^{2k}} \frac{|w_{k,j}| - 1}{2} \geqslant \frac{n - 2^{2k}}{2}. \tag{39}$$

Thus

$$|\Delta| \geqslant \sum_{\substack{k \leqslant \log n/2 \\ j \in \{1, ..., 2^{2k}\}}} |R'_{2k,j}| \geqslant \sum_{k=0}^{\lfloor \log n/2 \rfloor} \frac{n - 2^{2k}}{2}$$

$$\geqslant \frac{n(\lfloor \log n/2 \rfloor + 1)}{2} - \sum_{k=0}^{\lfloor \log n/2 \rfloor} 2^{2k} \geqslant \frac{n \log n - 3n}{4},$$

where the last step is justified by the inequality $\sum_{k=0}^{t} 2^{2k} \leqslant \frac{3}{2} 2^{2t}$, which can easily be proved by induction on t.

It remains to prove (a) and (b).

Proof of (a). Assume (p, q_{j+i2^k}, p') and $(p', a_{j+(i+1)2^k}, q)$ are any two consecutive transitions in $\rho_{k,j}$ (a successful run on $w_{k,j}$). From (36), we obtain $p \in Q_{\leqslant j+i2^k}$ and $q \in Q_{\geqslant j+(i+1)2^k+1}$. Consequently, the sum of the lengths of the two transitions is at least 2^k+1, which means the length of one of them is greater than 2^{k-1}.

Proof of (b). First of all, the claim is true when $|w_{k,j}| = 1$, because this implies immediately $2^{k+1} > n$, but n is the maximum length of any transition. So in the rest we assume $|w_{k,j}| > 1$.

Let $\tau = (p, a_{j+i2^k}, q)$ be an arbitrary transition in $\rho_{k,j}$. We distinguish several cases depending on the value of i.

If $i = 0$, we argue as follows. Let (q, a_{j+2^k}, q') be the transition right after τ in $\rho_{k,j}$. Then, by (36), $q \in Q_{\leqslant j+2^k}$. Since we assume $j \leqslant 2^k$, we get $q \in Q_{2^{k+1}-1}$; hence the length of τ is less than 2^{k+1}.

If $i = l_{k,j}$, the argument is similar. First, we know $j + (l_{k,j}+1)\,2^k > n$. Let $(p_0, a_{j+(l_{k,j}-1),p})$ be the transition right before τ in $\rho_{k,j}$. From (36), we obtain $p \in Q_{\geqslant j+(l_{k,j}-1)2^k}$, and so the length of τ is less than $(n+1) - (j + (l_{k,j}-1)\,2^k + 1) \leqslant (j + (l_{k,j}+1)\,2^k) - (j + (l_{k,j}-1)\,2^k + 1) = 2^{k+1} - 1$.

If $0 < i < l_{k,j}$, the argument goes as follows. Let $(p_0,\ a_{j+(i-1),p})$ and $(q,\ a_{j+(i+1)2^k},\ q')$ be the transitions right before, respectively right after, τ in $\rho_{k,j}$. By (36), we get $p \in Q_{\geqslant j+(i-1)2^k+1}$ and $q \in Q_{\leqslant j+(i+1)2^k}$, and so the length of τ is at most $2^{k+1}-1$. ∎

ACKNOWLEDGMENTS

We thank Volker Diekert for helpful comments on an early draft of this paper and Maxime Chrochemore for pointing us to relevant references.

REFERENCES

1. R. Book, S. Even, S. Greibach, and G. Ott, Ambiguity in graphs and expressions, *IEEE Trans. Comput.* **20** (1971), 149–153.
2. A. Brüggemann-Klein, Regular expressions into finite automata, *Theoret. Comput. Sci.* **120** (1993), 197–213.
3. M. Crochemore and C. Hancart, Automata for matching patterns, *in* "Handbook of Formal Languages, Vol. 2: Linear Modeling: Background and Application" (G. Rozenberg and A. Salomaa, Eds.), Chap. 9, pp. 399–462, Springer-Verlag, Berlin, 1997.
4. C.-H. Chang and R. Paige, From regular expressions to DFA's using compressed NFA's, *Theoret. Comput. Sci.* **178** (1997), 1–36.
5. A. Ehrenfeucht and P. Zeiger, Complexity measures for regular expressions, *J. Comput. System Sci.* **12** (1976), 134–146.
6. V. M. Glushkov, The abstract theory of automata, *Russian Math. Surveys* **16** (1961), 1–53. [Translation from *Usp. Mat. Naut.* **16** (1961), 3–41, by J. M. Jackson.]

7. C. Hagenah and A. Muscholl, Computing ε-free NFA from regular expressions in $\mathcal{O}(n \log^2(n))$ time, *in* "MFCS '98" (L. Prim *et al.*, Eds.), Springer Lecture Notes in computer Science, pp. 277–285, Springer-Verlag, Berlin, 1998.
8. J. E. Hopcroft and J. D. Ullman, "Introduction to Automata Theory, Languages and Computation," Addison–Wesley, Reading, MA, 1979.
9. J. Hromkovič, S. Seibert, and T. Wilke, Translating regular expression into small ε-free nondeterministic automata, *in* "STACS 97" (R. Reischuk and M. Morvan, Eds.), Vol. 1200 of Lecture Notes in Computer Science, pp. 55–66, Springer-Verlag, Berlin, 1997.
10. R. F. McNaughton and H. Yamada, Regular expressions and state graphs for automata, *IRE Trans. Electron. Comput.* **9** (1960), 39–47.
11. M. O. Rabin and D. Scott, Finite automata and their decision problems, *IBM J. Res. Develop.* **3** (1959), 114–125.
12. S. Sippu and E. Soisalon-Soininen, "Parsing Theory, Vol. I: Languages and Parsing," Vol. 15 of EATCS Monographs on Theoretical Computer Science, Springer-Verlag, Berlin, 1988.
13. K. Thompson, Regular expression search, *Commun. ACM* **11** (1968), 419–422.