Theoretical Computer Science 410 (2009) 3438–3446

Contents lists available at ScienceDirect

Theoretical Computer Science

journal homepage: www.elsevier.com/locate/tcs

Deciding determinism of caterpillar expressions[☆]

Kai Salomaa[a,*], Sheng Yu[b], Jinfeng Zan[a]

[a] School of Computing, Queen's University, Kingston, Ontario K7L 3N6, Canada

[b] Department of Computer Science, University of Western Ontario, London, Ontario N6A 5B7, Canada

ARTICLE INFO

Keywords:
Regular expressions
Tree-walking automata
Determinism
Decidability

ABSTRACT

Caterpillar expressions have been introduced by Brüggemann-Klein and Wood for applications in markup languages. Caterpillar expressions provide a convenient formalism for specifying the operation of tree-walking automata on unranked trees. Here we give a formal definition of determinism of caterpillar expressions that is based on the language of instruction sequences defined by the expression. We show that determinism of caterpillar expressions can be decided in polynomial time.

1. Introduction

Tree-walking automata have been used for the specification of context in structured documents and for tree pattern matching, for references see e.g. [24,25]. Differing from the classical tree automata, these applications typically use unranked trees where the number of children of a given node is finite but unbounded. In the unranked case, for example, when considering down moves of a tree-walking automaton the finite transition function cannot directly specify an arbitrary child node where the automaton moves to.

Brüggemann-Klein and Wood [7,8] introduced caterpillar expressions as a convenient tool to specify style sheets for XML documents. For possible applications of caterpillar expressions see also [13,23,26]. A caterpillar expression is, roughly speaking, a regular expression built from atomic instructions and such expressions provide an intuitive and simple formalism for specifying the operation of tree-walking automata on unranked trees. Each atomic instruction specifies the direction of the next move or a test on the current node label. The sequences of legal instructions define the computations of a tree-walking automaton on an unranked input tree. Given a caterpillar expression a crucial question is whether the computation it defines is deterministic. Recently Bojańczyk and Colcombet [2] have shown that nondeterministic tree-walking automata cannot, in general, be simulated by the deterministic variant. In their original paper Brüggemann-Klein and Wood discussed the notion of determinism only informally and presented examples of deterministic caterpillars.

Here we will give a formal definition of determinism of caterpillar expressions in terms of the set of instruction sequences defined by the expression. We show that determinism of caterpillar expressions can be decided in polynomial time. The general algorithm is based on ideas that have been used to test code properties of regular languages [1,19]. We develop a more direct algorithm to test determinism of caterpillar expressions where the corresponding instruction language has fixed polynomial density. The algorithm relies only on structural properties of the given caterpillar expressions. Regular languages having polynomial density have been characterized in terms of regular expressions without nested stars [30,31]. Also, we show that general caterpillar expressions have the same expressive power as nondeterministic tree-walking automata.

☆ A preliminary version of this paper appeared in the proceedings of CIAA 2007. Work supported in part by the Natural Sciences and Engineering Research Council of Canada grants OGP0147224 (Salomaa) and OGP0041630 (Yu).

* Corresponding author. Tel.: +1 613 533 6073; fax: +1 613 533 6513.
E-mail addresses: ksalomaa@cs.queensu.ca (K. Salomaa), syu@csd.uwo.ca (S. Yu).

0304-3975/$ – see front matter © 2009 Elsevier B.V. All rights reserved.
doi:10.1016/j.tcs.2009.03.010

To conclude the introduction we mention some recent results on tree-walking automata. It has been a long-standing open question whether tree-walking automata recognize all regular tree languages. A negative answer was conjectured by Engelfriet et al. [11,12] and Bojańczyk and Colcombet [3] have established this result. Neven and Schwentick [26] and Okhotin et al. [27] have investigated restricted classes of tree-walking automata and obtained negative recognizability results for these classes.

2. Preliminaries

We assume that the reader is familiar with the basic notions associated with regular expressions and finite automata [21, 31].

The set of words over an alphabet Ω is Ω^* and the empty word is λ. The length of a word $u \in \Omega^*$ is $|u|$. If u is nonempty, the first symbol of u is denoted $\text{first}(u)$. The prefix-relation for words over alphabet Ω is denoted $\leq_p$, that is, for $u, v \in \Omega^*$, $u \leq_p v$ if and only if $v = uu'$ for some $u' \in \Omega^*$. Similarly the "strict prefix" relation is denoted by $<_p$. We denote $u \simeq_p v$ if $u \leq_p v$ or $v \leq_p u$. The longest common prefix of words u and v is denoted as $\text{lcp}(u, v)$. The left-quotient of v by u, $u \setminus v$, is equal to w where $uw = v$ if $u \leq_p v$, and $u \setminus v$ is undefined otherwise. The *ordered symmetric difference* of words $u, v \in \Omega^*$, $u \triangle v$, is defined as follows:

$$u \triangle v = \begin{cases} ((u \setminus v), 1) \text{ if } u \leq_p v, \\ ((v \setminus u), 2) \text{ if } v <_p u, \\ \text{undefined otherwise.} \end{cases} \tag{1}$$

The second component of the value of $u \triangle v$ is used to indicate which of the words is a prefix of the other since the two cases are not symmetric.

A nondeterministic finite automaton (NFA) is a tuple $A = (\Omega, Q, q_0, F, \delta)$ where Ω is the input alphabet, Q is the finite set of states, $q_0 \in Q$ is the start state, $F \subseteq Q$ is the set of accepting states and $\delta \subseteq Q \times \Omega \times Q$ is the set of transitions. The language recognized by A is denoted $L(A) \subseteq \Omega^*$. The NFA A is said to be *reduced* if for any state $q \in Q$ there is a path from q_0 to q and a path from q to some accepting state. The NFA A a deterministic finite automaton (DFA) if for any $q \in Q$ and $b \in \Omega$ there exists at most one $q' \in Q$ such that $(q, b, q') \in \delta$.

The density function of a language $L \subseteq \Omega^*$ is defined as $\varrho_L(n) = |L \cap \Omega^n|$, $n \in \mathbb{N}$. We recall the following characterization of polynomial density regular languages from [30,31], similar results can be found also e.g. in [10].

Proposition 2.1. *A regular language R over Ω has density in $O(n^k)$, $k \geq 0$, iff R can be denoted by a finite union of regular expressions of the form*

$$w_0 u_1^* w_1 u_2^* \ldots u_{m+1}^* w_{m+1}, \quad m \leq k \tag{2}$$

where $w_i, u_j \in \Omega^*$, $i = 0, \ldots m+1$, $j = 1, \ldots m+1$.

We call finite unions of regular expressions as in (2), *k-bounded regular expressions over Ω*.

Below we still recall a few notions associated with trees and tree automata. General references for tree automata are [9, 14] and aspects specific to unranked trees are discussed e.g. in [5].

The set of positive integers is $\mathbb{N}$. In the following Σ denotes always a finite alphabet that is used to label the nodes of the trees. A *tree domain* D is a subset of $\mathbb{N}^*$ such that if $u \in D$ then every prefix of u is in D and there exists $m_u \geq 0$ such that for $j \in \mathbb{N}$, $u \cdot j \in D$ iff $j \leq m_u$. A Σ-labeled tree is a mapping $t : D \to \Sigma$ where $D = \text{dom(t)}$ is a tree domain. If Σ is a ranked alphabet, each symbol $\sigma \in \Sigma$ has a fixed rank denoted $\text{rank}(\sigma) \in \mathbb{N}$, and the rank determines the number of children of all nodes labeled by σ. In the general case, when referring to unranked trees, the label $t(u)$ of a node u does not specify the number of children of u, m_u (and there is no a priori upper bound for m_u). The set of Σ-labeled trees is denoted T_Σ.

3. Caterpillar expressions

Caterpillar expressions have been introduced in [7]. Here we present a somewhat streamlined definition that includes only what will be needed below for discussing determinism.

Definition 3.1. Let Σ be a set of node labels for the input trees. The set of atomic caterpillar instructions is

$$\Delta = \Sigma \cup \{\mathit{isFirst}, \mathit{isLast}, \mathit{isLeaf}, \mathit{isRoot}, \mathit{Up}, \mathit{Left}, \mathit{Right}, \mathit{First}, \mathit{Last}\}. \tag{3}$$

A *caterpillar expression* is a regular expression over Δ.

An atomic instruction $a \in \Sigma$ tests whether the label of the current node is a. The instructions *isFirst*, *isLast*, *isLeaf* and *isRoot* test whether the current node is the first (leftmost) sibling of its parent, the last sibling, a leaf node or the root node, respectively. The above are the *test instructions.*

The *move instructions Up*, *Left*, *Right*, *First* and *Last*, respectively, make the caterpillar move from the current node to its parent, the next sibling to the left, the next sibling to the right, the leftmost child of the current node, or the rightmost child of the current node, respectively.

Let α be a caterpillar expression. By the *instruction language of* α, $L(\alpha)$, we mean the set of all sequences of instructions over Δ that are denoted by the expression α (when α is viewed as an ordinary regular expression). Below we define the configurations and computation relation associated with an expression α. Intuitively, the computations can be viewed as a tree-walking automaton that, on an input tree t, implements all possible sequences of instructions in $L(\alpha)$.

Formally, a *t-configuration* of α is a pair (u, w) where $t \in T_\Sigma$ is the input tree, $u \in \mathrm{dom}(t)$ is the current node and and $w \in \Delta^*$ is the remaining sequence of instructions. The single step computation relation between t-configurations is defined by setting $(u, w) \vdash (u', w')$ if $w = cw'$, $c \in \Delta$, $w' \in \Delta^*$, $u, u' \in \mathrm{dom}(t)$, and the following holds:

(i) If c is a test instruction, c returns true at node $u \in \mathrm{dom}(t)$ and $u' = u$.
(ii) If c is one of the move instructions *Up*, *Left*, *Right*, *First* or *Last* then, respectively, $u = u'j$, $j \in \mathbb{N}$ (u' is the parent of u), $u = v(j+1)$, $u' = vj$, $v \in \mathbb{N}^*$, $j \in \mathbb{N}$ (u' is the sibling of u immediately to the left), $u = vj$, $u' = v(j+1)$, $v \in \mathbb{N}^*$, $j \in \mathbb{N}$ (u' is the sibling of u immediately to the right), $u' = u1$ (u' is the leftmost child of u), or $u' = uj$, $j \in \mathbb{N}$ and $u(j+1) \notin \mathrm{dom}(t)$ (u' is the rightmost child of u).

Let α be a caterpillar expression. The tree language defined by α is

$$T(\alpha) = \{\ t \in T_\Sigma \ \mid\ (\exists w \in L(\alpha))\ \ (\lambda, w) \vdash^* (u, \lambda) \text{ for some } u \in \mathrm{dom}(t)\ \}.$$

Thus $t \in T(\alpha)$ if and only if some sequence of instructions denoted by α can be executed to completion where the computation begins at the root of t and ends at an arbitrary node of t. The definition could alternatively require that the caterpillar has to return to the root of t at the end of the computation.

Example 3.1. Let $a, b \in \Sigma$. Define α as the expression

$$(\mathit{First} \cdot \mathit{Right}^*)^* \cdot \mathit{isFirst} \cdot (\mathit{isLeaf} \cdot a \cdot \mathit{Right})(\mathit{isLeaf} \cdot b \cdot \mathit{Right})(\mathit{isLeaf} \cdot a \cdot \mathit{isLast}).$$

The caterpillar α defines the set of trees that contain a node with precisely three children that are all leaves and labeled, respectively, by a, b, a.

The behaviour of a caterpillar expression is described using a tree-walking automaton and, conversely, we show that caterpillar expressions can simulate arbitrary tree-walking automata. We state the result below comparing the expressive power of caterpillar expressions and tree-walking automata only for tree languages over a ranked alphabet. Most of the work on tree-walking automata, e.g., [2,12,26], uses trees over ranked alphabets.

Theorem 3.1. *Let Σ be a ranked alphabet. Caterpillar expressions define the same sets of Σ-labeled trees as the nondeterministic tree-walking automata.*

Proof. We need to show only how to simulate a tree-walking automaton A by a caterpillar expression. We denote the set of states of A as Q and m is the maximum rank of elements of Σ. The transitions of A are defined as a set of tuples (q, σ, j, q'), where $q \in Q$ is the current state, $\sigma \in \Sigma$ is the current node label, $j \in \{0, 1, \ldots, \mathrm{rank}(\sigma)\}$ is the direction of the next move and $q' \in Q$ is the state after the move. Here "0" is an up move and "i", $1 \le i \le \mathrm{rank}(\sigma)$, denotes a move to the ith child.

Denote $\Omega = Q \times \Sigma \times \{0, 1, \ldots, m\} \times Q$. The set of *semi-computations* of A is the regular language $L_{sc} \subseteq \Omega^*$ that consists of all words $\omega_1 \cdots \omega_k$, where $\omega_i \in \Omega$ is a tuple that represents a transition of A, $i = 1, \ldots, k$, and $\pi_1(\omega_1)$ is the start state of A, $\pi_4(\omega_k)$ is an accepting state of A and $\pi_4(\omega_i) = \pi_1(\omega_{i+1})$, $i = 1, \ldots, k-1$. Here π_j is the projection to the jth component.

Any accepting computation of A corresponds to a word of L_{sc} but, conversely, words of L_{sc} need not represent an accepting computation since the definition of L_{sc} requires only that the computation is locally correct and does not verify that the number of up moves does not exceed the number of down moves. However, the language L_{sc} will give the following correspondence with instruction languages defined by caterpillar expressions.

Let Δ be as in (3) and define a mapping $f : \Omega^* \to \Delta^*$ by setting

$$f(q, \sigma, j, q') = \begin{cases} \sigma \cdot \mathit{First} \cdot (\mathit{Right})^{j-1} \text{ if } 1 \le j \le \mathrm{rank}(\sigma), \\ \sigma \cdot \mathit{Up} \text{ if } j = 0. \end{cases} \tag{4}$$

Now the language $f(L_{sc})$ is regular and hence it is denoted by some caterpillar expression α_{sc}. The instruction sequences of $L(\alpha_{sc})$ correspond to semi-computations of A where we have deleted the state information, and any $u \in L(\alpha_{sc})$ can be completed to a semi-computation according to the correspondence (4). As observed above, a semi-computation need not represent a correct computation of A due to the possibility of trying to make an up move at the root of the tree. In this situation also the execution of the corresponding sequence of caterpillar instructions obtained via the function f gets blocked. This means that $w \in L_{sc}$ encodes a valid computation on $t \in T_\Sigma$ iff the sequence of caterpillar instructions $f(w)$ can be successfully executed on t. Hence $T(\alpha_{sc})$ is exactly the tree language recognized by A. ■

The result of Theorem 3.1 can be straightforwardly extended for unranked trees assuming we extend the operation of tree-walking automata to unranked trees in some reasonable way, e.g., the down moves could be made only to the first or last child and then the automaton could make moves to the closest sibling node. The proof of Theorem 3.1 did not use several of the caterpillar instructions. For example, the test *isLeaf* is not needed because on ranked trees this property can be decided by looking at the node label. Similarly, (deterministic) tree-walking automata on unranked trees would need a mechanism to detect whether the node is a leaf. For unranked trees the details of the simulation would depend on the precise definition of the tree-walking automaton model.

4. Formal definition of determinism

By definition, a caterpillar expression can be simulated by a tree-walking automaton [8,12,27] and, intuitively, we say that a caterpillar is deterministic if the computation performing the simulation is deterministic. This operational definition was used by Brüggemann-Klein and Wood [7,8] to deal with the notion of determinism. However, for example in order to algorithmically decide determinism of given caterpillar expressions, it is necessary to have a more direct definition of determinism in terms of the sequences of instructions denoted by a caterpillar expression.

Let Δ be the set of atomic instructions given in Definition 3.1. Let $t \in T_\Sigma$ be arbitrary. We say that instruction $c \in \Delta$ is *successfully executed* at node $u \in \mathrm{dom}(t)$ if there exist $w \in \Delta^*$ and $u' \in \mathrm{dom}(t)$ such that $(u, cw) \vdash (u', w)$. (Without loss of generality we could choose w to be λ.)

Definition 4.1. Let $c, c' \in \Delta$. We say that instructions c and c' are *mutually exclusive* if either

(i) $c, c' \in \Sigma$ and $c \neq c'$, that is, c and c' are tests on distinct symbols of Σ, or,
(ii) $\{c, c'\}$ is one of the sets $\{First, isLeaf\}$, $\{Last, isLeaf\}$, $\{Up, isRoot\}$, $\{Left, isFirst\}$, or $\{Right, isLast\}$.

The following lemma is verified by a straightforward case analysis.

Lemma 4.1. *For any $c, c' \in \Delta$, $c \neq c'$, the following two conditions are equivalent.*

(i) *There exists $t \in T_\Sigma$ and $u \in \mathrm{dom}(t)$ such that c and c' can be successfully executed at node u.*
(ii) *The instructions c and c' are not mutually exclusive.*

In order for a caterpillar expression α to define a deterministic computation, we require that in computations controlled by α on any input tree there cannot be a situation where the computation could successfully execute two different instructions as the next step. Formally, we define the notion of determinism associated with caterpillar expressions as follows.

Definition 4.2. Let α be a caterpillar expression over Δ. We say that α is *deterministic* if the following implication holds. If wc_1w_1 and wc_2w_2 are in $L(\alpha)$ where $w, w_1, w_2 \in \Delta^*$, $c_1, c_2 \in \Delta$, $c_1 \neq c_2$, then c_1 and c_2 are mutually exclusive.

The definition says that for any instruction sequences w and w' defined by α that are not prefixes of one another, the pair of instructions following the longest common prefix of w and w' has to be mutually exclusive. Note that if $w, w' \in L(\alpha)$ where w is a proper prefix of w', this corresponds to a situation where the corresponding tree-walking automaton has reached an accepting state after simulating the instructions of w and the tree-walking automaton can execute further moves. According to our definition this does not constitute an instance of nondeterminism. By Lemma 4.1, Definition 4.2 coincides with the operational definition of determinism discussed earlier.

Note that the condition of Definition 4.2, strictly speaking, depends only on the instruction language of α. In the following, when there is no confusion, we say that a language $L \subseteq \Delta^*$ is deterministic if L satisfies the condition of Definition 4.2.

The caterpillar of Example 3.1 is obviously nondeterministic. The subexpression $(First \cdot Right^*)^*$ involves choices between instructions *First* and *Right*, and these allow the caterpillar to move from the root to an arbitrary node.

Example 4.1. The below construction is modified from [7].

$$\begin{aligned}\alpha_{\mathrm{trav}} = {} & First^* \cdot isLeaf \cdot (Right \cdot First^* \cdot isLeaf)^* \cdot isLast \cdot \\ & (Up \cdot (Right \cdot First^* \cdot isLeaf)^* \cdot isLast)^* \cdot isRoot.\end{aligned}$$

In computations defined by α_{trav}, the subexpression $First^* \cdot isLeaf$ finds the leftmost leaf of the tree. Next the subexpression $(Right \cdot First^* \cdot isLeaf)^* \cdot isLast$ finds the leftmost leaf of the current subtree that is the last child of its parent. The process is then iterated by going one step up in the subexpression $(Up \cdot \ldots \cdot isLast)^*$ and in this way it can be verified that the expression α_{trav} defines a computation that traverses an arbitrary input tree in depth-first left-to-right order. Furthermore, it is easy to verify that α_{trav} is deterministic. In the notations of Definition 4.2 possible pairs of instructions c_1, c_2 that may occur in the instruction sequences are $\{First, isLeaf\}$, $\{Right, isLast\}$ and $\{Up, isRoot\}$ and these are all mutually exclusive.

Concerning other similar notions, we note that it might seem that determinism of caterpillar expressions is related to unambiguity of regular expressions [4,6]. However, it is not difficult to verify that deterministic expressions need not be unambiguous or vice versa. Also, we can note that determinism of caterpillar expressions is not the same as the notion of determinism of generalized finite automata where the transitions may be labeled by regular expressions [17,20].

In Theorem 3.1 we have seen that general caterpillar expressions can simulate nondeterministic tree-walking automata. The morphism f used in the proof of Theorem 3.1, roughly speaking, erases the state information from encodings of (semi-)computations and the instruction language ($\subseteq \Delta^*$) corresponding to a deterministic tree-walking automaton need not be deterministic in the sense of Definition 4.2. On the other hand, the deterministic caterpillar expression considered in Example 4.1 can traverse an arbitrary input tree which indicates that it may not be very easy to show that some particular tree language (recognized by a deterministic tree-walking automaton) cannot be defined by any deterministic caterpillar expression.

Open problem 4.1. *Do the deterministic tree-walking automata define a strictly larger family of tree languages than the tree languages defined by deterministic caterpillar expressions?*

5. Deciding determinism

We show that determinism can be decided in polynomial time for general caterpillar expressions. Given a caterpillar expression α it would not be difficult to verify whether or not α satisfies the condition of Definition 4.2 assuming we can construct the minimal DFA for the instruction language of α. However, this approach would result in an exponential time algorithm due to the exponential worst-case blow-up of converting a regular expression to a DFA. As a side remark we mention that, interestingly, it was established recently [16] that also the converse holds: there exist DFAs defined over a fixed alphabet that are exponentially more succinct than any equivalent regular expressions.

First we give an algorithm to test determinism that is based on the state-pair graph associated with a reduced NFA recognizing the instruction language of α. The construction relies on ideas that have been used to test code properties of regular languages [1,19].

Definition 5.1. Let $A = (\Omega, Q, q_0, F, \delta)$ be an NFA. The state-pair graph of A is defined as a directed graph $G_A = (V, E)$ where the set of nodes is $V = Q \times Q$ and the set of Ω-labeled edges is

$$E = \{((p, q), b, (p', q')) \mid (p, b, p') \in \delta,\ (q, b, q') \in \delta,\ b \in \Omega\}.$$

Lemma 5.1. *Assume that $A = (\Delta, Q, q_0, F, \delta)$ is a reduced NFA with input alphabet Δ as in (3). The language $L(A)$ is not deterministic if and only if there exist $p, q \in Q$ such that*

(i) *The state-pair graph G_A has a path from (q_0, q_0) to (p, q).*
(ii) *There exist $c_1, c_2 \in \Delta$, $c_1 \neq c_2$, such that $(p, c_1, p') \in \delta$ and $(q, c_2, q') \in \delta$ for some $p', q' \in Q$, and c_1, c_2 are not mutually exclusive.*

Proof. First assume that $L(A)$ is not deterministic in the sense of Definition 4.2. Thus, there exist $w, w_1, w_2 \in \Delta^*$, $c_1, c_2 \in \Delta$, $c_1 \neq c_2$, where c_1 and c_2 are not mutually exclusive, such that $wc_iw_i \in L(A)$, $i = 1, 2$. Let C_i be an accepting computation of A on the word wc_iw_i, and let p_i be the state of C_i after reading the prefix w. This means that in the graph G_A the node (p_1, p_2) is reachable from (q_0, q_0) and a transition on c_i is defined in state p_i, $i = 1, 2$. Thus, the conditions (i) and (ii) hold.

Conversely, assume that $p, q, p', q' \in Q$ and $c_1, c_2 \in \Delta$ are as in (i) and (ii). Since G_A has a path from (q_0, q_0) to (p, q) there exists $w \in \Delta^*$ such that both p and q are reachable from q_0 on word w. Since A is reduced, there exists $w_{p'}$ (respectively, $w_{q'}$) that reaches an accepting state from p' (respectively, q'). Thus $wc_1w_{p'}, wc_2w_{q'} \in L(A)$ and $L(A)$ is not deterministic. ∎

In the second part of the proof, note that we require that (p, q) is reachable from (q_0, q_0) in the graph G_A whereas the accepting states can be reached from p' and q' along computations of A not necessarily along the same word.

Lemma 5.2. *Given a caterpillar expression α of size n over an alphabet Δ as in (3) we can construct in time $O(n^2 \log^4 n)$ the state-pair graph G_A of an NFA A that recognizes the instruction language $L(\alpha)$ of α.*

Proof. For α having size n we can construct an NFA (without ε-transitions) with $O(n \cdot (\log n)^2)$ transitions and the transformation can be done in time $O(n \log n + m)$ where m is the size of the output [18,22,28]. The NFA can be reduced and the corresponding state-pair graph can be constructed in square time in the size of the NFA. Here the size of the NFA refers to the sum of the number of states and the number of transitions. ∎

Note that if Δ is considered to be fixed, the upper bound for the regular expression-to-NFA conversion can be improved [15,28]. Combining the results of Lemmas 5.1 and 5.2 with any graph reachability algorithm we have:

Theorem 5.1. *Given an alphabet Δ as in (3) and a caterpillar expression α over Δ we can decide in polynomial time whether or not α is deterministic.*

The algorithm of Theorem 5.1 relies on state-pair graphs and other notions that have been introduced to test code properties of regular languages. To conclude this section we develop an algorithm to decide determinism that relies only on structural properties of the caterpillar expression given as input. This algorithm is restricted to k-bounded caterpillar expressions, that is, expressions where the instruction language has polynomial density.

Let Δ be as in (3). In the following $k \in \mathbb{N}$ is fixed and we consider caterpillar expressions that are sums of expressions of the form

$$x_0 y_1^* x_1 y_2^* x_2 \cdots y_{m+1}^* x_{m+1}, \qquad x_i, y_j \in \Delta^*, \quad y_j \neq \lambda, \tag{5}$$

$i = 0, \ldots, m+1, j = 1, \ldots, m+1, m \leq k$. Note that above the assumption $y_j \neq \lambda$ can be made without loss of generality. If α is as above, by the length of α we mean $|\alpha| = |x_0| + \sum_{i=1}^{m+1} |x_i y_i|$.

We say that an expression (5) is *normalized* if

$$\text{for each } 1 \leq i \leq m+1, \quad \mathrm{lcp}(x_i, y_i) = \lambda, \quad \text{and,} \tag{6}$$

$$x_j \neq \lambda, \quad \text{for each } 1 \leq j \leq m. \tag{7}$$

We begin with the following technical lemma.

Lemma 5.3. *Consider an arbitrary expression α of length n as in (5) and let k be the constant bounding m. The expression α can be written as the sum of $O(n^k)$ normalized expressions each having length $O(k \cdot n)$.*

Proof. Corresponding to a subexpression $y_i^* x_i$ of α as in (5), we can find an equivalent 'left-shifted'' expression

$$\mathrm{LS}(y_i^* x_i) = z(y_i')^* x_i' \tag{8}$$

where $\mathrm{lcp}(y_i', x_i') = \lambda$. If y_i is not a prefix of x_i we denote $\mathrm{lcp}(y_i, x_i) = z$ and choose $z(y'z)^*x'$ as the right-hand side of (8), where $y_i = zy'$, $x_i = zx'$. If y_i is a prefix of x_i (i.e., above $y' = \lambda$), we first write $y_i^* x_i$ as zz^*x' and then apply the process iteratively to the expression z^*x'.

More generally, if α is as in (5) we define the left-shifted expression $\mathrm{LS}(\alpha)$ to be the expression obtained from α by applying this operation iteratively from right to left. That is, first $y_{m+1}^* x_{m+1}$ is replaced by $\mathrm{LS}(y_{m+1}^* x_{m+1}) = z_{m+1}(y_{m+1}')^* x_{m+1}'$, then $y_m^*(x_m z_{m+1})$ is replaced by $\mathrm{LS}(y_m^*(x_m z_{m+1}))$, and so on. The left-shift operation eliminates from α subexpressions $y_i^* x_i$ where y_i and x_i have a nonempty common prefix, that is, for $\mathrm{LS}(\alpha)$ the condition (6) holds. Note that $\mathrm{LS}(\alpha)$ will be of the form (5) since in (8) $y_i' \neq \lambda$ always when $y_i \neq \lambda$.

Using the left-shift operation we below define an inductive process to rewrite α as a sum of normalized expressions. At the end of the proof we describe the upper bound estimates for the number and the size of the components in the sum.

By applying the left-shift operation we can guarantee that α as in (5) satisfies the property (6). Let $1 \leq i \leq m$ be the largest index such that $x_i = \lambda$. (Note that condition (7) only tries to prevent "consecutive stars" and (7) allows the possibility that $x_{m+1} = \lambda$.) We call i the largest index of consecutive stars and proceed by induction on i. Denote $\beta = x_0 y_1^* \cdots y_{i-1}^* x_{i-1}$, $\gamma = x_{i+1} y_{i+2}^* \cdots y_{m+1}^* x_{m+1}$.

If $y_i = y_{i+1}$, we can write $\alpha = \beta y_i^* y_{i+1}^* \gamma$ simply as $\beta y_i^* \gamma$. This expression satisfies (6) and the largest index of consecutive stars in $\beta y_i^* \gamma$ is strictly less than i. In the following we can assume that $y_i \neq y_{i+1}$. We consider separately the cases $y_i \not\asymp_p y_{i+1}$, $y_i <_p y_{i+1}$ and $y_{i+1} <_p y_i$. In the first two cases we write

$$\alpha = \beta y_i^* y_{i+1}^* \gamma = \beta y_i^* \gamma + \beta y_i^* y_{i+1} y_{i+1}^* \gamma. \tag{9}$$

The expression $\mathrm{LS}(\beta y_i^* \gamma)$ satisfies (6), and when $\mathrm{LS}(\beta y_i^* \gamma)$ is written in form (5), the largest index of consecutive stars is strictly less than i.

In the following we show how to handle the expression $\delta = \beta y_i^* y_{i+1} y_{i+1}^* \gamma$.

(i) $y_i \not\asymp_p y_{i+1}$: Now $y_i = zy_i'$, $y_{i+1} = zy_{i+1}'$, $y_i', y_{i+1}' \neq \lambda$, $\mathrm{lcp}(y_i', y_{i+1}') = \lambda$. Thus we can write δ in the form $\beta z(y_i' z)^* y_{i+1}' y_{i+1}^* \gamma$ and the above expression satisfies (6) and there the largest index of consecutive stars is strictly less than i.

(ii) $y_i <_p y_{i+1}$: We write $y_i = z_1 z_2$, $z_2 \neq \lambda$ where $y_{i+1} = (z_1 z_2)^r z_1 z_3$, $r \geq 1$, $\mathrm{lcp}(z_2, z_3) = \lambda$. That is, y_{i+1} has a prefix consisting of r copies of y_i and z_1 is the longest common prefix of the remaining suffix of y_{i+1} and y_i.

In this case δ can be written in the equivalent form

$$\beta(z_1 z_2)^r z_1 (z_2 z_1)^* z_3 y_{i+1}^* \gamma. \tag{10}$$

Here we have two subcases. (a) Assume that $z_3 \neq \lambda$. Now since $z_2 \neq \lambda$, and $\mathrm{lcp}(z_2, z_3) = \lambda$, it follows that applying the left-shift operation to (10) we have reduced the largest index of consecutive stars. (The left-shift operation would change only the "prefix" $\beta(z_1 z_2)^r z_1$ of the expression (10).)

(b) Secondly, we consider the case $z_3 = \lambda$. Now if $z_1 z_2 = z_2 z_1$, then by the Lyndon-Schützenberger theorem [29], z_1 and z_2 are both powers of some word v, and hence we can write also $y_i = v^t$, $y_{i+1} = v^s$, for some $t, s \geq 0$. This means that in the original expression α we can replace $y_i^* y_{i+1}^*$ by $(v^{z_1} + \cdots + v^{z_h})(v^{t+s})^*$ where $0 \leq z_1 < \cdots < z_h < t + s$.

In the following we then assume that $z_1 z_2 \neq z_2 z_1$. In this case we write the expression (10) (and remembering $z_3 = \lambda$, $y_{i+1} = (z_1 z_2)^r z_1$) as

$$\beta(z_1 z_2)^r z_1 (z_2 z_1)^* \gamma + \beta(z_1 z_2)^r z_1 (z_2 z_1)^* (z_1 z_2)^r z_1 y_{i+1}^* \gamma.$$

In the first expression of the sum we have reduced the number of stars. In the second expression the largest index of consecutive stars remains i. Since $z_2 z_1 \not\asymp_p (z_1 z_2)^r z_1$, the second expression is of the type handled in case (i) above.

(iii) Finally we consider the case $y_{i+1} <_p y_i$. Symmetrically to the above case (ii) we can now write

$$y_{i+1} = z_1 z_2, \qquad z_2 \neq \lambda, \qquad y_i = (z_1 z_2)^r z_1 z_3, \qquad \mathrm{lcp}(z_2, z_3) = \lambda. \tag{11}$$

Instead of (9) we write

$$\alpha = \beta y_i^* \gamma + \beta y_i^* y_{i+1} \gamma + \cdots + \beta y_i^* y_{i+1}^r \gamma + \beta y_i^* y_{i+1}^{r+1} y_{i+1}^* \gamma. \tag{12}$$

In the first $r + 1$ terms appearing on the right-hand side of (12) we have reduced the total number of stars. Hence applying the left-shift operation produces an expression that satisfies (6) where the largest index of consecutive stars is strictly less than i.

It is sufficient to consider the last expression in the sum on the right side of (12). When substituting the notations (11) this expression becomes

$$\beta((z_1 z_2)^r z_1 z_3)^* (z_1 z_2)^{r+1} (z_1 z_2)^* \gamma = \beta(z_1 z_2)^r z_1 (z_3 (z_1 z_2)^r z_1)^* z_2 (z_1 z_2)^* \gamma. \tag{13}$$

If $z_3 \neq \lambda$, and recalling that $z_2 \neq \lambda$, $\mathrm{lcp}(z_2, z_3) = \lambda$, the right-hand side of (13) satisfies (6) and there we have reduced the largest index of consecutive stars.

It remains to consider the case $z_3 = \lambda$. If $z_1z_2 = z_2z_1$, the words z_1 and z_2 are powers of the same word and the expression is handled exactly as in (ii) above. Assume then that $z_1z_2 \neq z_2z_1$. Now the right-hand side of (13) can be written as

$$\beta(z_1z_2)^r z_1((z_1z_2)^r z_1)^* z_2\gamma + \beta(z_1z_2)^r z_1((z_1z_2)^r z_1)^* z_2(z_1z_2)(z_1z_2)^*\gamma.$$

In the first expression we have reduced the largest index of consecutive stars. Since $(z_1z_2)^r z_1 \not\simeq_p z_2z_1$, the second expression of the sum is of the type handled in case (i) above.

The value of i is bounded by k and, at each stage, the inductive process branches into two subexpressions except that in (12) we branch into $r+2$ subexpressions, where $r \in O(n)$. Thus, $O(n^k)$ is a very rough upper bound for the total number of expressions. Each stage of the inductive process increases the length of the expression at most by adding a new factor y_i or y_{i+1}. Hence the size of each of the resulting expressions is bounded by $O(k \cdot n)$. ■

Let α be as in (5). We say that α is *well-behaved* if $x_i \neq \lambda$ implies that first(y_i) and first(x_i) are mutually exclusive, $1 \leq i \leq m+1$.

Note that always $y_i \neq \lambda$. If α is normalized, then x_i can be the empty word only when $i = m+1$. When considering prefixes of $L(\alpha)$, where α is normalized, after the last symbol of y_i the next symbol can be one of first(y_i) and first(x_i) and these are known to be distinct. Hence the following lemma is immediate.

Lemma 5.4. *If α as in (5) is normalized and deterministic, then α is well-behaved.*

Due to Lemmas 5.3 and 5.4, in order to test determinism of k-bounded expressions it is sufficient to consider sums of well-behaved normalized expressions. Consider two well-behaved normalized k-bounded expressions over Δ,

$$\alpha = x_0y_1^*x_1 \cdots y_{m+1}^*x_{m+1}, \qquad \beta = u_0v_1^*u_1 \cdots v_{q+1}^*u_{q+1}, \quad m, q \leq k. \tag{14}$$

We describe an algorithm `TestNormalizedExpr` to test whether or not $\alpha + \beta$ is deterministic where α and β are as in (14). The algorithm determines whether there exist a prefix of $L(\alpha)$, w_α, and a prefix of $L(\beta)$, w_β, such that

$$w_\alpha \quad \text{and} \quad w_\beta \text{ violate the condition of determinism.} \tag{15}$$

We introduce the following notation:

(i) $Y(i_1, \ldots, i_r) = x_0y_1^{i_1}x_1y_2^{i_2} \cdots x_{r-1}y_r^{i_r}$, $1 \leq r \leq m+1, i_b \geq 0, b = 1, \ldots, r$.
(ii) $V(j_1, \ldots, j_s) = u_0v_1^{j_1}u_1v_2^{j_2} \cdots u_{s-1}y_s^{j_s}$, $1 \leq s \leq q+1, j_b \geq 0, b = 1, \ldots, s$.

A word $Y(i_1, \ldots, i_r)$ (respectively, $V(j_1, \ldots, j_s)$) is a prefix of a word in $L(\alpha)$ (respectively, in $L(\beta)$). Note that if $r < m+1$ then $Y(i_1, \ldots, i_r, 0)$ equals to $Y(i_1, \ldots, i_r)x_{i_r}$ and the words $V(j_1, \ldots, j_s)$ satisfy an analogous property.

We say that the *index* of a pair of words $(Y(i_1, \ldots, i_r), V(j_1, \ldots, j_s))$ is (r, s). Note that since α is normalized, always when $r \neq r'$ we have $Y(i_1, \ldots, i_r) \neq Y(i'_1, \ldots, i'_{r'})$ independently of the parameters $i_1, \ldots, i_r$ and $i'_1, \ldots, i'_{r'}$. The V-words have the analogous property since β is normalized and this means that the index of a pair of words is uniquely defined.

The algorithm uses a method Compare(w_1, w_2), for prefixes w_1 of $L(\alpha)$ and prefixes w_2 of $L(\beta)$. The method finds the longest common prefix of w_1 and w_2 and looks at the following symbols of w_1 and w_2. Only in the case where w_1 is a prefix of w_2 or vice versa, Compare(w_1, w_2) does not directly give an answer, and the algorithm has to continue comparing possible continuations of w_1 and w_2.

The algorithm begins by comparing $Y(0) = x_0$ and $V(0) = u_0$. For the general case, we consider a method call

$$\text{Compare}(Y(i_1, \ldots, i_r), V(j_1, \ldots j_s)), \quad r, s \geq 1. \tag{16}$$

of the recursive algorithm. The recursive calls to the compare method (16) are defined according to the following 4 cases.

(i) If $Y(i_1, \ldots, i_r) \not\simeq_p V(j_1, \ldots j_s)$, then the method call (16) gives a definitive answer for this branch. By looking at the symbols following the longest common prefix we either have found an instance of nondeterminism or no continuation of the words $Y(i_1, \ldots, i_r)$ and $V(j_1, \ldots j_s)$ can violate the condition of determinism.

(ii) Consider now the case $Y(i_1, \ldots, i_r) <_p V(j_1, \ldots j_s)$. The algorithm is looking for words as in (15) and hence next it should expand the shorter word $Y(i_1, \ldots, i_r)$ by appending a word y_r or x_r. Hence the algorithm makes two recursive calls:
Compare$(Y(i_1, \ldots, i_r + 1), V(j_1, \ldots j_s))$, Compare$(Y(i_1, \ldots, i_r, 0), V(j_1, \ldots j_s))$.
If $r = m+1$, above the word

$$Y(i_1, \ldots, i_r, 0) \quad \text{is interpreted as } Y(i_1, \ldots, i_r) \cdot x_{m+1}. \tag{17}$$

Note that since α is normalized, $y_r, x_r \neq \lambda$ and first$(x_r) \neq$ first(y_r). Hence for at most one word $X \in \{Y(i_1, \ldots, i_r + 1), Y(i_1, \ldots, i_r, 0)\}, X \simeq_p V(j_1, \ldots j_s)$. This means that at least one of the above recursive calls gives the answer directly and the algorithm needs to continue only (at most) one path.

(iii) The case where $V(j_1, \ldots j_s) <_p Y(i_1, \ldots, i_r)$ is symmetric to the above.

(iv) Finally, consider the case where $Y(i_1, \ldots, i_r) = V(j_1, \ldots j_s)$. Now when trying to find words as in (15), the algorithm may expand $Y(i_1, \ldots, i_r)$ by appending a word x_r or y_r, and expand $V(j_1, \ldots j_s)$ by appending a word u_s or v_s. This leads to four recursive calls:
Compare$(Y(i_1, \ldots, i_r + 1), V(j_1, \ldots j_s + 1))$,
Compare$(Y(i_1, \ldots, i_r + 1), V(j_1, \ldots j_s, 0))$,
Compare$(Y(i_1, \ldots, i_r, 0), V(j_1, \ldots j_s + 1))$,
and Compare$(Y(i_1, \ldots, i_r, 0), V(j_1, \ldots j_s, 0))$.
In cases where $r = m+1$ or $s = q+1$ we make notational conventions analogous to (17). Since α and β are normalized, at most two of the four pairs of words in the arguments are in the prefix-relation with each other, and in the other cases the compare method gives directly an answer. However, here the algorithm may need to branch into two independent computations.

Now we establish a time bound for the algorithm. The essential idea is that the algorithm employs a counter that keeps track of the number of recursive calls (16) that have taken place since the index of the pair of argument words was changed. We claim that the number of consecutive method calls (16) where the arguments have index (r, s) can be bounded by $|y_r| + |v_s|$.

To prove the claim, consider a sequence of $|y_r|+|v_s|$ method calls where the index of the pair of argument words remains (r, s). Thus there must exist $0 \le b \le b'$, $0 \le c \le c'$, $(b, c) \ne (b', c')$, such that our sequence makes recursive calls Compare$(Y(i_1, \ldots, i_r + b), V(j_1, \ldots, j_s + c)$ and Compare$(Y(i_1, \ldots, i_r + b'), V(j_1, \ldots, j_s + c')$ where

$$Y(i_1, \ldots, i_r + b) \triangle V(j_1, \ldots, j_s + c) = Y(i_1, \ldots, i_r + b') \triangle V(j_1, \ldots, j_s + c').$$

Here $\triangle$ is the ordered symmetric difference of words defined in (1). The above equality follows from the pigeon-hole principle when considering the different positions where the end point of $Y(i_1, \ldots, i_r+b)$ (respectively, of $V(j_1, \ldots, j_s+c)$) may "hit" the current last occurrence of v_s (respectively, of y_r). Note that the equality of the ordered symmetric differences entails that if on the left-hand side the Y-word is longer than the V-word, the same holds on the right-hand side, and vice versa. Thus the equality means that this branch of the computation is in a cycle and will never find words as in (15). This concludes the proof of the claim.

Thus we have seen that one branch of the recursive calls (16) uses linear time as a function of the argument word lengths. The length of the arguments of (16) can be longer than the input length $n = |\alpha|+|\beta|$. We have observed that in the arguments of (16) we can always restrict $i_r, j_s \le |y_r|+|v_s|$ and hence $|Y(i_1, \ldots, i_r)|, |V(j_1, \ldots, j_s)| \in O(n^2)$. Strictly speaking, according to the above description, one branch of the computation makes $O(n)$ calls to the compare method, but by keeping track of the current positions in prefixes of $L(\alpha)$ and $L(\beta)$ the total time of one branch of the computation can also be bounded by $O(n^2)$.

The recursive calls (16) may branch into two according to case (iv). With a fixed index (r, s) this branching needs to be done at most once. Putting all the above together (and relying on Lemma 5.4) we have the following lemma.

Lemma 5.5. *On two normalized k-bounded expressions of length n the algorithm* `TestNormalizedExpr` *operates in time* $2^{2k} \cdot O(n^2)$.

Combining Lemmas 5.3 and 5.5 we get the following:

Proposition 5.1. *Let Δ be as in (3) and k is fixed. Given a k-bounded caterpillar expression α over Δ (i.e., α is a sum of arbitrarily many expressions as in (5)) an algorithm based on Lemma 5.3 and* `TestNormalizedExpr` *decides in polynomial time whether or not α is deterministic.*

Note that the time bound of `TestNormalizedExpr` to decide determinism of a sum of normalized expressions is of the form $f(k)O(n^2)$. In the time bound the function f depends exponentially on k but, since the branching occurs only when the current prefixes coincide, in fact, on most inputs the running time should be essentially better. Arbitrary k-bounded expressions need to be written as sums of normalized expressions (according to Lemma 5.3) and the worst-case behaviour of the algorithm of Proposition 5.1 would not be better than the behaviour of the general algorithm of Theorem 5.1. The algorithm of Proposition 5.1 relies only on structural properties of the inputs and it may be useful in cases where the input expressions are to begin with in a normalized form.

References

[1] J. Berstel, D. Perrin, Theory of Codes, Academic Press, Inc., 1985.
[2] M. Bojańczyk, T. Colcombet, Tree walking automata cannot be determinized, Theoret. Comput. Sci. 350 (2006) 164–173.
[3] M. Bojańczyk, T. Colcombet, Tree-walking automata do not recognize all regular languages, SIAM J. Comput. 38 (2008) 658–701.
[4] R.V. Book, S. Even, S. Greibach, G. Ott, Ambiguity in graphs and expressions, IEEE Trans. Computers 20 (1971) 149–153.
[5] A. Brüggemann-Klein, M. Murata, D. Wood, Regular tree and regular hedge languages over unranked alphabets, Technical Report HKUST-TCSC-2001-0, The Hongkong University of Science and Technology, 2001.
[6] A. Brüggemann-Klein, D. Wood, One-unambiguous regular languages, Inform. Computation 142 (1998) 182–206.
[7] A. Brüggemann-Klein, D. Wood, Caterpillars: A context-specification technique, Mark-up Languages: Theory & Practice 2 (2000) 81–106.
[8] A. Brüggemann-Klein, D. Wood, Caterpillars, context, tree automata and tree pattern matching, in: G. Rozenberg, W. Thomas (Eds.), Developments in Language Theory, DLT'99, World Scientific, 2000, pp. 270–285.

[9] H. Comon, R. Gilleron, F. Jacquemard, D. Lugiez, S. Tison, M. Tommasi, Tree automata techniques and applications, 2002. Available at: http://www.grappa.univ-lille3.fr/tata.

[10] S. Eilenberg, Automata, Languages, and Machines, vol. A., Academic Press, New York, 1974.

[11] J. Engelfriet, H.J. Hoogeboom, Tree-walking pebble automata, in: J. Karhumäki, H. Maurer, G. Păun, G. Rozenberg (Eds.), Jewels are forever, Springer-Verlag, 1999, pp. 72–83.

[12] J. Engelfriet, H.J. Hoogeboom, J.P. van Best, Trips on trees, Acta Cybern. 14 (1999) 51–64.

[13] H. Fernau, Learning XML grammars, in: Machine Learning and Data Mining in Pattern Recognition, MLDM'01, in: Lect. Notes Comput. Sci., vol. 2123, Springer, 2001, pp. 73–87.

[14] F. Gécseg, M. Steinby, Tree languages, in: G. Rozenberg, A. Salomaa (Eds.), Handbook of Formal Languages, vol. 3, Springer, 1997, pp. 1–68.

[15] V. Geffert, Translation of binary regular expressions into nondeterministic ε-free automata with $O(n \log n)$ transitions, J. Comput. System Sci. 66 (2003) 451–472.

[16] W. Gelade, F. Neven, Succinctness of the complement and intersection of regular expressions, in: S. Albers, P. Weil (Eds.), Proceedings of STACS'08, 2008, pp. 325–336.

[17] D. Giammarresi, R. Montalbano, Deterministic generalized automata, Theoret. Comput. Sci. 215 (1999) 191–208.

[18] C. Hagenah, A. Muscholl, Computing ε-free NFA from regular expressions in $O(n \log^2(n))$ times, RAIRO Theoret. Inform. Appl. 34 (2000) 257–277.

[19] Y.-S. Han, K. Salomaa, D. Wood, Intercode regular languages, Fund. Inform. 76 (2007) 113–128.

[20] Y.-S. Han, D. Wood, The generalization of generalized automata: Expression automata, Int. J. Found. Comput. Sci. 16 (2005) 499–510.

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

[22] J. Hromkovič, S. Seibert, T. Wilke, Translating regular expressions into small ε-free nondeterministic automata, J. Comput. System Sci. 62 (2001) 565–588.

[23] P. Kilpeläinen, D. Wood, SGML and XML document grammars and exceptions, Inform. and Comput. 163 (2001) 230–251.

[24] T. Milo, D. Suciu, V. Vianu, Typechecking for XML transformers, J. Comput. System Sci. 66 (2002) 66–97.

[25] M. Murata, D. Lee, M. Mani, Taxonomy of XML schema languages using formal language theory, ACM Trans. Internet Technol. 5 (2005).

[26] F. Neven, T. Schwentick, On the power of tree walking automata, Inform. and Comput. 183 (2003) 86–103.

[27] A. Okhotin, K. Salomaa, M. Domaratzki, One-visit caterpillar tree automata, Fund. Inform. 52 (2002) 361–375.

[28] G. Schnitger, Regular expressions and NFA without ε-transitions, in: B. Durand, W. Thomas (Eds.), Proceedings of STACS'06, in: Lect. Notes Comput. Sci., vol.3884, 2006, pp. 432–443.

[29] H.J. Shyr, Free Monoids and Languages, 3rd ed., Hon Min Book Company, Taichung, 2001.

[30] A. Szilard, S. Yu, K. Zhang, J. Shallit, Characterizing regular languages with polynomial densities, in: Proc. of 17th MFCS, in: Lect. Notes Comput. Sci., vol. 629, Springer-Verlag, 1992, pp. 494–503.

[31] S. Yu, Regular languages, in: G. Rozenberg, A. Salomaa (Eds.), Handbook of Formal Languages, vol. 1, Springer-Verlag, 1997, pp. 41–110.