Deterministic Regular Expressions with Back-References

Dominik D. Freydenberger[*1] **and Markus L. Schmid**[2]

1 **University of Bayreuth, Bayreuth, Germany**
ddfy@ddfy.de
2 **University of Trier, Trier, Germany**
MSchmid@uni-trier.de

Abstract

Most modern libraries for regular expression matching allow back-references (i. e., repetition operators) that substantially increase expressive power, but also lead to intractability. In order to find a better balance between expressiveness and tractability, we combine these with the notion of determinism for regular expressions used in XML DTDs and XML Schema. This includes the definition of a suitable automaton model, and a generalization of the Glushkov construction.

1998 ACM Subject Classification F.1.1 Models of Computation, F.4.3 Formal Languages

Keywords and phrases Deterministic Regular Expression, Regex, Glushkov Automaton

Digital Object Identifier 10.4230/LIPIcs.STACS.2017.33

1 Introduction

Regular expressions were introduced in 1956 by Kleene [26] and quickly found wide use in both theoretical and applied computer science. While the theoretical interpretation of regular expressions remains mostly unchanged (as expressions that describe exactly the class of regular languages), modern applications use variants that vary greatly in expressive power and algorithmic properties. This paper tries to find common ground between two of these variants with opposing approaches to the balance between expressive power and tractability.

The first variant that we consider are *regex*, regular expressions that are extended with a *back-reference operator*. This operator is used in almost all modern programming languages (like e. g. Java, PERL, and .NET). For example, the regex $\langle x\colon (\mathtt{a} \vee \mathtt{b})^* \rangle \cdot \&x$ defines $\{ww \mid w \in \{\mathtt{a}, \mathtt{b}\}^*\}$, as $(\mathtt{a} \vee \mathtt{b})^*$ can create a $w \in \{\mathtt{a}, \mathtt{b}\}^*$, which is then stored in the variable x and repeated with the reference $\&x$. Hence, back-references allow to define non-regular languages; but with the side effect that the membership problem is NP-complete (cf. Aho [2]).

The other variant, *deterministic regular expressions* (also known as *1-unambiguous regular expressions*), uses an opposite approach, and achieves a more efficient membership problem than regular expressions by defining only a strict subclass of the regular languages.

Intuitively, a regular expression is deterministic if, when matching a word from left to right with no lookahead, it is always clear where in the expression the next symbol must be matched. This property has a characterization via the *Glushkov construction* that converts every regular expression α into a (potentially non-deterministic) finite automaton $\mathcal{M}(\alpha)$, by treating each terminal position in α as a state. Then α is deterministic if $\mathcal{M}(\alpha)$ is deterministic. As a consequence, the membership problem for deterministic regular expressions can be solved

* Dominik D. Freydenberger was supported by DFG grant FR 3551/1-1.

more efficiently than for regular expressions in general (more details can be found in [24]). Hence, in spite of their limited expressive power, deterministic regular expressions are used in actual applications: Originally defined for the ISO standard for SGML (see Brüggemann-Klein and Wood [9]), they are a central part of the W3C recommendations on XML DTDs [7] and XML Schema [22] (see Murata et al. [32]).

The goal of this paper is finding common ground between these two variants, by introducing *deterministic regex* and an appropriate automaton model, the *deterministic memory automata with trap-state* (DTMFA). To elaborate: We first introduce a new automaton model for regex, the memory automata with trap-state (TMFA). While the TMFA is based on the MFA that was proposed by Schmid [35], its deterministic variant, the DTMFA, is better suited for complementation than the deterministic MFA. We then generalize the notion of deterministic regular expressions to regex, and show that the Glushkov construction can also be generalized. This allows us not only to efficiently decide the membership problem for deterministic regex, but also whether a regex is deterministic. After this, we study the expressive power of these models. Although deterministic regex share many of the limitations of deterministic regular expressions (in particular, the inherent non-determinism of some regular languages persists), their expressive power offers some surprises. Finally, we examine a subclass of deterministic regexes and DTMFA for which polynomial space minimization is possible, and we consider an alternative notion of determinism.

From the perspective of deterministic regular expressions, this paper proposes a natural extension that significantly increases the expressive power, while still having a tractable membership problem. From a regex point of view, we restrict regex to their deterministic core, thus obtaining a tractable subclass. Hence, the authors intend this paper as a starting point for further work, as it opens a new direction on research into making regex tractable. For space reasons, detailed proofs are given in a full version of the paper [21].

Main contributions. The main conceptual contribution of this paper are the notion of determinism in regex, and an appropriate deterministic automaton model. The main challenge from this point of view was finding a natural extension of deterministic regular expressions that preserves the following properties: A natural definition of determinism that can be checked efficiently and also has an automata-theoretic characterization, and an efficient Glushkov-style conversion to automata that decide the membership problem efficiently. Regarding technical contributions, the authors would like to emphasize that, in addition to the effort that was needed to accomplish the aforementioned goals, the paper uses subtleties of the back-reference operator in novel ways. By using these, deterministic regex can define non-deterministic regular languages (in particular, all unary regular languages), as well as infinite languages that are not pumpable in the usual sense.

Related work. Regex were first examined from a theoretical point of view by Aho [2], but without fully defining the semantics. There were various proposals for semantics, of which we mention the first by Câmpeanu, Salomaa, Yu [10], and the recent one by Schmid [35], which is the basis for this paper. Apart from defining the semantics, there was work on the expressive power [10, 11, 20], the static analysis [11, 18, 19], and the tractability of the membership problem (investigated in terms of a strongly restricted subclass of regex) [16, 17]. They have also been compared to related models in database theory, e. g. graph databases [4] and information extraction [15, 19].

Following the original paper by Brüggemann-Klein and Wood [9], deterministic regular expressions have been studied extensively. Aspects include computing the Glushkov au-

tomaton and deciding the membership problem (e. g. [8, 24, 34]), static analysis (cf. [31]), deciding whether a regular language is deterministic (e. g. [12, 24, 30]), closure properties and descriptional complexity [28], and learning (e. g. [5]). One noteworthy extension are counter operators (e. g. [23, 24, 27]), which we briefly address in Section 7.

2 Preliminaries

We use ε to denote the *empty word*. The subset and proper subset relation are denoted by $\subseteq$ and $\subset$, respectively. Let Σ be a finite terminal alphabet. Unless otherwise noted, we assume $|\Sigma| \geq 2$. Let Ξ be an infinite variable alphabet with $\Xi \cap \Sigma = \emptyset$. Let $w \in \Sigma^*$, then, for every i, $1 \leq i \leq |w|$, $w[i]$ denotes the symbol at position i of w. We define $w^0 := \varepsilon$ and $w^{i+1} := w^i \cdot w$ for all $i \geq 0$, and, for $w = a_1 \cdots a_n$ with $a_i \in \Sigma$, let $w^{m+\frac{i}{n}} = w^m \cdot a_1 \cdots a_i$ for all $m \geq 0$ and all i with $0 \leq i \leq n$. A $v \in \Sigma^*$ is a *factor* of w if there exist $u_1, u_2 \in \Sigma^*$ with $w = u_1 v u_2$. If $u_2 = \varepsilon$, v is also a *prefix* of w.

We use the notions of deterministic and non-deterministic finite automata (DFA and NFA) like [25]. If an NFA can have ε-transitions, we call it an ε-NFA. Given a class $\mathcal{C}$ of language description mechanisms (e. g., a class of automata or regular expressions), we use $\mathcal{L}(\mathcal{C})$ to denote the class of all languages $\mathcal{L}(C)$ with $C \in \mathcal{C}$. The *membership problem for* $\mathcal{C}$ is defined as follows: Given a $C \in \mathcal{C}$ and a $w \in \Sigma^*$, is $w \in \mathcal{L}(C)$?

2.1 Regex

▶ **Definition 1** (Syntax of regex)**.** We define RX, the set of *regex* over Σ and Ξ, recursively:
Terminals and ε: $a \in \mathsf{RX}$ and $\mathsf{var}(a) = \emptyset$ for every $a \in (\Sigma \cup \{\varepsilon\})$.
Variable reference: $\&x \in \mathsf{RX}$ and $\mathsf{var}(\&x) = \{x\}$ for every $x \in \Xi$.
Concatenation: $(\alpha \cdot \beta) \in \mathsf{RX}$ and $\mathsf{var}(\alpha \cdot \beta) = \mathsf{var}(\alpha) \cup \mathsf{var}(\beta)$ if $\alpha, \beta \in \mathsf{RX}$.
Disjunction: $(\alpha \vee \beta) \in \mathsf{RX}$ and $\mathsf{var}(\alpha \vee \beta) = \mathsf{var}(\alpha) \cup \mathsf{var}(\beta)$ if $\alpha, \beta \in \mathsf{RX}$.
Kleene plus: $(\alpha^+) \in \mathsf{RX}$ and $\mathsf{var}(\alpha^+) = \mathsf{var}(\alpha)$ if $\alpha \in \mathsf{RX}$.
Variable binding: $\langle x\colon \alpha\rangle \in \mathsf{RX}$ and $\mathsf{var}(\langle x\colon \alpha\rangle) = \mathsf{var}(\alpha) \cup \{x\}$ if $\alpha \in \mathsf{RX}$ with $x \in \Xi \setminus \mathsf{var}(\alpha)$.
In addition, we allow $\emptyset$ as a regex (with $\mathsf{var}(\emptyset) = \emptyset$), but we do not allow $\emptyset$ to occur in any other regex. An $\alpha \in \mathsf{RX}$ with $\mathsf{var}(\alpha) = \emptyset$ is called a *proper regular expression*, or just *regular expression*. We use REG to denote the set of all regular expressions.

We add and omit parentheses freely, as long as the meaning remains clear. We use the Kleene star α^* as shorthand for $\varepsilon \vee \alpha^+$, and A as shorthand for $\bigvee_{a \in A} a$ for non-empty $A \subseteq \Sigma$.

We define the semantics of regex using the *ref-words* (short for *reference words*) by Schmid [35]. A ref-word is a word over $(\Sigma \cup \Xi \cup \Gamma)$, where $\Gamma := \{[_x,]_x, \mid x \in \Xi\}$. Intuitively, the symbols $[_x$ and $]_x$ mark the beginning and the end of the match that is stored in the variable x, while an occurrence of x represents a reference to that variable. Instead of defining the language of a regex α directly, we first treat α as a generator of ref-words by defining its *ref-language* $\mathcal{R}(\alpha)$. If $\alpha \in \Sigma \cup \{\varepsilon\}$, $\mathcal{R}(\alpha) := \{\alpha\}$; and $\mathcal{R}(\&x) := \{x\}$ for all $x \in \Xi$. Furthermore, $\mathcal{R}(\alpha \cdot \beta) := \mathcal{R}(\alpha) \cdot \mathcal{R}(\beta)$, $\mathcal{R}(\alpha \vee \beta) := \mathcal{R}(\alpha) \cup \mathcal{R}(\beta)$, and $\mathcal{R}(\alpha^+) := \mathcal{R}(\alpha)^+$. Finally, $\mathcal{R}(\langle x\colon \alpha\rangle) := ([_x \mathcal{R}(\alpha)]_x)$. For regular expressions, $\mathcal{L}(\alpha) = \mathcal{R}(\alpha)$. Alternatively, $\mathcal{R}(\alpha) := \mathcal{L}(\alpha_{\mathcal{R}})$, where the proper regular expression $\alpha_{\mathcal{R}}$ is obtained by replacing each sub-regex $\langle x\colon \beta\rangle$ of α with $[_x \beta_{\mathcal{R}}]_x$, and each $\&x$ with x.

Intuitively speaking, every occurrence of a variable x in some $r \in \mathcal{R}(\alpha)$ functions as a pointer to the next factor $[_x v]_x$ to the left of this occurrence (or to ε if no such factor exists). In this way, a ref-word r compresses a word over Σ, the so-called dereference $\mathcal{D}(r)$ of r, which can be obtained by replacing every variable occurrence x by the corresponding factor v (note

that v might again contain variable occurrences, which need to be replaced as well), and removing all symbols $[_x,]_x \in \Gamma$ afterwards. See [35] for a more detailed definition, or the following Example 2 for an illustration. Finally, we define $\mathcal{L}(\alpha) := \{\mathcal{D}(r) \mid r \in \mathcal{R}(\alpha)\}$.

▶ **Example 2.** Let $\alpha := \big(\langle x\colon (\mathtt{a} \vee \mathtt{b})^+\rangle \& x\big)^+$. Then $\mathcal{R}(\alpha) = \{[_x w_1]_x \cdot x \cdots [_x w_n]_x \cdot x \mid n \geq 1, w_i \in \{\mathtt{a},\mathtt{b}\}^+\}$. Hence, $\mathcal{L}(\alpha) = (L_{\mathsf{copy}})^+$, with $L_{\mathsf{copy}} := \{ww \mid w \in \{\mathtt{a},\mathtt{b}\}^+\}$. Let $\alpha_{\mathsf{sq}} := \big(\langle x\colon \& y\rangle\langle y\colon \& x \cdot \mathtt{a}\rangle\big)^*$. Then $\mathcal{R}(\alpha_{\mathsf{sq}}) = \{\big([_x y]_x \cdot [_y x \cdot \mathtt{a}]_y\big)^i \mid i \geq 0\}$. For example, consider the ref word $r_3 = [_x y]_x \cdot [_y x \cdot \mathtt{a}]_y \cdot [_x y]_x \cdot [_y x \cdot \mathtt{a}]_y \cdot [_x y]_x \cdot [_y x \cdot \mathtt{a}]_y$ with $\mathcal{D}(r_3) = \mathtt{a}^9$. Using induction, we can verify that $\mathcal{D}(r_i) = \mathtt{a}^{i^2}$. Thus, $\mathcal{L}(\alpha_{\mathsf{sq}}) = \{\mathtt{a}^{n^2} \mid n \geq 0\}$.

Hence, unlike regular expressions, regex can define non-regular languages. The expressive power comes at a price: their membership problem is NP-complete (follows from Angluin [3]), and various other problems are undecidable (Freydenberger [18]). Starting with Aho [2], there have been various approaches to specifying syntax and semantics of regex. While [2] only sketched the intuition behind the semantics, the first formal definition (using parse trees) was proposed by Câmpeanu, Salomaa, Yu [10], followed by the ref-words of Schmid [35]. For a comparison between these approaches and actual implementations, see the full version [21].

3 Memory Automata with Trap State

Memory automata [35] are a simple automaton model that characterizes $\mathcal{L}(\mathsf{RX})$. Intuitively speaking, these are classical finite automata that can record consumed factors in memories, which can be recalled later on in order to consume the same factor again. However, for our applications, we need to slightly adapt this model to *memory automata with trap-state*.

▶ **Definition 3.** For every $k \in \mathbb{N}$, a *k-memory automaton with trap-state*, denoted by $\mathsf{TMFA}(k)$, is a tuple $M = (Q, \Sigma, \delta, q_0, F)$, where Q is a finite set of *states* that contains the *trap-state* $[\mathsf{trap}]$, Σ is a finite *alphabet*, $q_0 \in Q$ is the *initial state*, $F \subseteq Q$ is the set of *final states* and $\delta\colon Q \times (\Sigma \cup \{\varepsilon\} \cup \{1, 2, \ldots, k\}) \to \mathcal{P}(Q \times \{\mathtt{o}, \mathtt{c}, \mathtt{r}, \diamond\}^k)$ is the *transition function* (where $\mathcal{P}(A)$ denotes the power set of a set A), which satisfies $\delta([\mathsf{trap}], b) = \{([\mathsf{trap}], \diamond, \diamond, \ldots, \diamond)\}$, for every $b \in \Sigma \cup \{\varepsilon\}$, and $\delta([\mathsf{trap}], i) = \emptyset$, for every i, $1 \leq i \leq k$. The elements $\mathtt{o}$, $\mathtt{c}$, $\mathtt{r}$ and $\diamond$ are called *memory instructions* (they stand for opening, closing and reseting a memory, respectively, and $\diamond$ leaves the memory unchanged).

A *configuration* of M is a tuple $(q, w, (u_1, r_1), \ldots, (u_k, r_k))$, where $q \in Q$ is the *current state*, w is the *remaining input* and, for every i, $1 \leq i \leq k$, (u_i, r_i) is the *configuration of memory i*, where $u_i \in \Sigma^*$ is the *content of memory i* and $r_i \in \{\mathtt{O}, \mathtt{C}\}$ is the *status of memory i* (i. e., $r_i = \mathtt{O}$ means that memory i is open and $r_i = \mathtt{C}$ means that it is closed). The *initial configuration* of M (*on input w*) is the configuration $(q_0, w, (\varepsilon, \mathtt{C}), \ldots, (\varepsilon, \mathtt{C}))$, a configuration $(q, w, (u_1, r_1), \ldots, (u_k, r_k))$ is an *accepting configuration* if $w = \varepsilon$ and $q \in F$.

M can change from a configuration $c = (q, vw, (u_1, r_1), \ldots, (u_k, r_k))$ to a configuration $c' = (p, w, (u'_1, r'_1), \ldots, (u'_k, r'_k))$, denoted by $c \vdash_M c'$, if there exists a transition $\delta(q, b) \ni (p, s_1, \ldots, s_k)$ with either ($b \in (\Sigma \cup \{\varepsilon\})$ and $v = b$) or ($b \in \{1, 2, \ldots, k\}$, $s_b = \mathtt{c}$ and $v = u_b$), and, for every i, $1 \leq i \leq k$,

$$
\begin{array}{llllll}
s_i = \diamond \wedge r_i = \mathtt{O} & \Rightarrow & (u'_i, r'_i) = (u_i v, r_i), & s_i = \diamond \wedge r_i = \mathtt{C} & \Rightarrow & (u'_i, r'_i) = (u_i, r_i), \\
s_i = \mathtt{o} & \Rightarrow & (u'_i, r'_i) = (v, \mathtt{O}), & s_i = \mathtt{c} & \Rightarrow & (u'_i, r'_i) = (u_i, \mathtt{C}), \\
s_i = \mathtt{r} & \Rightarrow & (u'_i, r'_i) = (\varepsilon, \mathtt{C}). & & &
\end{array}
$$

Furthermore, M can change from a configuration $(q, vw, (u_1, r_1), \ldots, (u_k, r_k))$ to the configuration $([\mathsf{trap}], w, (u_1, r_1), \ldots, (u_k, r_k))$, if $\delta(q, b) \ni (p, s_1, \ldots, s_k)$ for some $p \in Q$, $b \in \{1, 2, \ldots, k\}$ and $s_b = \mathtt{c}$, such that $u_b = vv'$ with $v' \neq \varepsilon$ and $v'[1] \neq w[1]$.

A transition $\delta(q, b) \ni (p, s_1, s_2, \ldots, s_k)$ is an *ε-transition* if $b = \varepsilon$ and is called *consuming*, otherwise (if all transitions are consuming, then M is called *ε-free*). If $b \in \{1, 2, \ldots, k\}$, it is called a *memory recall transition* and the situation that a memory recall transition leads to the state [trap], is called a *memory recall failure*.

The symbol $\vdash_M^*$ denotes the reflexive and transitive closure of $\vdash_M$. A $w \in \Sigma^*$ is *accepted* by M if $c_{\text{init}} \vdash_M^* c_f$, where c_{init} is the initial configuration of M on w and c_f is an accepting configuration. The set of words accepted by M is denoted by $\mathcal{L}(M)$.

Note that executing the open action o on a memory that already contains some word discards the previous contents of that memory. For illustrations and examples for TMFA, we refer to [35]. A crucial part of TMFA is the trap-state [trap], in which computations terminate, if a memory recall failure happens. If [trap] is not accepting, then TMFA are (apart from negligible formal differences) identical to the memory automata introduced in [35], which characterize the class of regex language. If, on the other hand, [trap] is accepting, then every computation with a memory recall failure is accepting (independent from the remaining input). While it seems counter-intuitive to define the words of a language via "failed" back-references, the possibility of having an accepting trap-state yields closure under complement for *deterministic* TMFA (see Theorem 6). It will be convenient to consider the partition of TMFA into $\mathsf{TMFA}^{\mathsf{rej}}$ and $\mathsf{TMFA}^{\mathsf{acc}}$ (having a rejecting and an accepting trap-state, respectively).

Every $\mathsf{TMFA}^{\mathsf{acc}}$ can be transformed into an equivalent $\mathsf{TMFA}^{\mathsf{rej}}$, which implies $\mathcal{L}(\mathsf{TMFA}) = \mathcal{L}(\mathsf{TMFA}^{\mathsf{rej}})$; thus, it follows from [35] that TMFA characterize $\mathcal{L}(\mathsf{RX})$. The idea of this construction is as follows. Every memory i is simulated by two memories $(i, 1)$ and $(i, 2)$, which store a (nondeterministically guessed) factorisation of the content of memory i. This allows us to guess and verify if a memory recall failure occurs, i. e., $(i, 1)$ stores the longest prefix that can be matched and $(i, 2)$ starts with the first mismatch. For correctness, it is crucial that every possible factorisation of the content of a memory i can be guessed.

▶ **Theorem 4.** $\mathcal{L}(\mathsf{TMFA}) = \mathcal{L}(\mathsf{TMFA}^{\mathsf{rej}}) = \mathcal{L}(\mathsf{RX})$.

A consequence of the proof is that TMFA inherits the NP-hardness of the membership problem from RX. We do not devote more attention to this, as we focus on deterministic TMFA: A TMFA is *deterministic* (or a DTMFA, for short) if δ satisfies $|\delta(q, b)| \leq 1$, for every $q \in Q$ and $b \in \Sigma \cup \{\varepsilon\} \cup \{1, 2, \ldots, k\}$ (for the sake of convenience, we then interpret δ as a partial function with range $Q \times \{\mathtt{o}, \mathtt{c}, \mathtt{r}, \diamond\}^k$), and, furthermore, for every $q \in Q$, if $\delta(q, x)$ is defined for some $x \in \{1, 2, \ldots, k\} \cup \{\varepsilon\}$, then, for every $y \in (\Sigma \cup \{\varepsilon\} \cup \{1, 2, \ldots, k\}) \setminus \{x\}$, $\delta(q, y)$ is undefined. Analogously to TMFA, we partition DTMFA into $\mathsf{DTMFA}^{\mathsf{acc}}$ and $\mathsf{DTMFA}^{\mathsf{rej}}$.

The algorithmically most important feature of DTMFA is that their membership can be solved efficiently by running the automaton on the input word. However, for each processed input symbol, there might be a delay of at most $|Q|$ steps, due to ε-transitions and recalls of empty memories, which leads to $O(|Q||w|)$. Removing such non-consuming transitions first, is possible, but problematic. In particular, recalls of empty memories depend on the specific input word and could only be determined beforehand by storing for each memory whether it is empty, which is too expensive. However, by $O(|Q|^2)$ preprocessing, we can compute the information that is needed in order to determine in $O(k)$ where to jump if certain memories are empty, and which memories are currently empty can be determined on-the-fly while processing the input. This leads to a delay of only k, the number of memories:

▶ **Theorem 5.** *Given $M \in$ DTMFA with n states and k memories, and $w \in \Sigma^*$, we can decide in time $O(n^2 + k|w|)$, whether or not $w \in \mathcal{L}(M)$.*

Note that the preprocessing in the proof of Theorem 5 is only required once, so we can solve the membership for several words w_i in $O(n^2 + k\sum|w_i|)$. Moreover, if it is guaranteed that no empty memories are recalled, then membership can be solved in $O(n + |w|)$ (where $O(n)$ is needed in order to remove ε-transitions).

Similar to DFA, it is possible to complement DTMFA by toggling the acceptance of states. However, for DTMFA, we have to remove ε-transitions and recalls of empty memories. In particular, the construction for Theorem 6 uses the finite control to store whether memories are empty or not, which causes a blow-up that is exponential in the number of memories.

▶ **Theorem 6.** $\mathcal{L}(\mathsf{DTMFA})$ *is closed under complement.*

We next discuss expressive power: If there is a constant upper bound on the lengths of contents of memories that are recalled in accepting computations of an $M \in \mathsf{DTMFA}$, then memories can be simulated by the finite state control; thus, $\mathcal{L}(M) \in \mathcal{L}(\mathsf{REG})$. Consequently, if $\mathcal{L}(M) \notin \mathcal{L}(\mathsf{REG})$, there is a word uvw that is accepted by recalling some memory with an arbitrarily large content v. Moreover, if $[\mathsf{trap}]$ is non-accepting, then no word can be accepted that contains u as a prefix, but not uv, since this will cause a memory recall failure. Intuitively speaking, a $\mathsf{DTMFA}^{\mathsf{rej}}$ for a non-regular language makes arbitrarily large "jumps":

▶ **Lemma 7** (Jumping Lemma)**.** *Let $L \in \mathcal{L}(\mathsf{DTMFA}^{\mathsf{rej}})$. Then either L is regular, or for every $m \geq 0$, there exist $n \geq m$ and $p_n, v_n \in \Sigma^+$ such that*
1. *$|v_n| = n$,*
2. *v_n is a factor of p_n,*
3. *$p_n v_n$ is a prefix of a word from L,*
4. *for all $u \in \Sigma^+$, $p_n u \in L$ only if v_n is a prefix of u.*

▶ **Example 8.** Let $L := \{ww \mid w \in \Sigma^*\}$ with $|\Sigma| \geq 2$, which is well-known to be not regular. Assume $L \in \mathcal{L}(\mathsf{DTMFA}^{\mathsf{rej}})$ and choose $m := 1$. Then there exist $n \geq 1$ and $p_n, v_n \in \Sigma^*$ that satisfy the conditions of Lemma 7. Choose $a \in \Sigma$ that is not the first letter of v_n, and define $u := ap_n a$. Then v_n is not a prefix of u, but $p_n u = (p_n a)^2 \in L$, which is a contradiction.

▶ **Example 9.** Let $L := \{\mathtt{a}^i\mathtt{b}\mathtt{a}^j \mid i > j \geq 0\}$. Using textbook methods, it is easily shown that L is not regular. Now, assuming that $L \in \mathcal{L}(\mathsf{DTMFA}^{\mathsf{rej}})$, choose $m := 4$. Then there exist $n \geq 4$ and $p_n, v_n \in \Sigma^+$ that satisfy the conditions of Lemma 7. As $p_n v_n$ is a prefix of a word in L, either $p_n = \mathtt{a}^i$ or $p_n = \mathtt{a}^i\mathtt{b}\mathtt{a}^j$ with $i, j \geq 0$ (and $i \geq 4$ or $i + j \geq 3$). In the first case, consider $u := \mathtt{b}\mathtt{a}$. Then $p_n u = \mathtt{a}^i\mathtt{b}\mathtt{a}$ with $i \geq 4$; hence, $p_n u \in L$. But u starts with $\mathtt{b}$, and v_n is a factor of $p_n = \mathtt{a}^i$. Contradiction, as v_n cannot be a prefix of u. For the second case, let $u := \mathtt{a}$. As $p_n v_n$ is a prefix of a word in L, and as $|v_n| = n$, $i > j + n \geq j + 4$ must hold. Hence, $p_n u = \mathtt{a}^i\mathtt{b}\mathtt{a}^{j+1}$, and $p_n u \in L$. Contradiction, as v_n is not a prefix of u.

For unary languages, there is an alternative to Lemma 7 that is easier to apply and characterizes unary $\mathsf{DTMFA}^{\mathsf{rej}}$-languages. It is built on the following definition: A language $L \subseteq \{\mathtt{a}\}^*$ is an *infinite arithmetic progression* if $L = \{\mathtt{a}^{bi+c} \mid i \geq 0\}$ for some $b \geq 1$, $c \geq 0$.

▶ **Lemma 10.** *Let $L \in \mathcal{L}(\mathsf{DTMFA}^{\mathsf{rej}})$ be an infinite language with $L \subseteq \{\mathtt{a}\}^*$. The following conditions are equivalent:*
1. *L is regular.*
2. *L contains an infinite arithmetic progression.*
3. *There is $b \geq 1$ such that, for every $n \geq 0$, $\mathtt{a}^{bi+c_n} \in L$ for some $c_n \geq 0$ and all $0 \leq i \leq n$.*

▶ **Example 11.** Let $\alpha := \langle x \colon \mathtt{a}\mathtt{a}^+\rangle(\&x)^+$ (this regex is also known as "Abigail's expression" [1] in the PERL community). Then $\mathcal{L}(\alpha) = \{\mathtt{a}^{mn} \mid m, n \geq 2\}$. In other words, α generates the language of all $\mathtt{a}^i$ such that i is a composite number (i. e., not a prime number). As $\mathcal{L}(\alpha)$ is not regular and contains the arithmetic progression $2i + 4$, Lemma 10 yields $\mathcal{L}(\alpha) \notin \mathcal{L}(\mathsf{DTMFA}^{\mathsf{rej}})$.

The following result is a curious consequence of Lemma 10:

▶ **Proposition 12.** *Over unary alphabets,* $\mathcal{L}(\mathsf{DTMFA}^{\mathsf{rej}}) \cap \mathcal{L}(\mathsf{DTMFA}^{\mathsf{acc}}) = \mathcal{L}(\mathsf{REG})$.

4 Deterministic Regex

In order to define deterministic regex as an extension of deterministic regular expressions, we first extend the notion of a *marked alphabet* that is commonly used for the latter: For every alphabet A, let $\tilde{A} := \{a_{(n)} \mid a \in A, n \geq 1\}$. For every $\alpha \in \mathsf{RX}$, we define $\tilde{\alpha}$ as a regex that is obtained by taking $\alpha_{\mathcal{R}}$ (the proper regular expression over $\Sigma \cup \Xi \cup \Gamma$ that generates the ref-language $\mathcal{R}(\alpha)$), and marking each occurrence of $\chi \in (\Sigma \cup \Xi \cup \Gamma)$ by a unique number (to make this well-defined, we assume that the markings start at 1 and are increased stepwise). For example, if $\alpha := \langle y \colon (\mathsf{a} \vee \&x)^* \cdot (\varepsilon \vee \mathsf{b} \cdot \mathsf{a}) \rangle \cdot \&y$, then $\tilde{\alpha} = [_{y(1)}(\mathsf{a}_{(2)} \vee x_{(3)})^* \cdot (\varepsilon \vee \mathsf{b}_{(4)} \cdot \mathsf{a}_{(5)})]_{y(6)} \cdot y_{(7)}$. We also use these markings in the ref-words: For example, $[_{y(1)}\mathsf{a}_{(2)}\mathsf{a}_{(2)}x_{(3)}\mathsf{a}_{(2)}]_{y(6)}y_{(7)} \in \mathcal{R}(\tilde{\alpha})$.

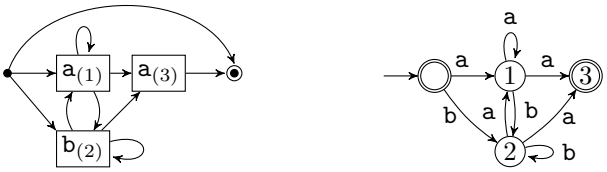
Before we explain this definition and use it to define deterministic regex, we first discuss the special case of deterministic regular expressions: A proper regular expression α is *not* deterministic if there exist words $u, v_1, v_2 \in \tilde{\Sigma}^*$, a terminal $a \in \Sigma$ and positions $i \neq j$ such that $ua_{(i)}v_1$ and $ua_{(j)}v_2$ are elements of $\mathcal{L}(\tilde{\alpha})$ (see e. g. [9, 24]). Otherwise, it is a *deterministic proper regular expression* (or, for short, just *deterministic regular expression*).

The intuition behind this definition is based on the Glushkov construction for the conversion of regular expressions into finite automata, as a regular expression α is deterministic if and only if its *Glushkov automaton* $\mathcal{M}(\alpha)$ is deterministic. Given a regular expression α, we define $\mathcal{M}(\alpha)$ in the following way: First, we use the marked regular expression $\tilde{\alpha}$ to construct its *occurrence graph*[1] $G_{\tilde{\alpha}}$, a directed graph that has a source node src, a sink node snk, and one node for each $a_{(i)}$ in $\tilde{\alpha}$. The edges are constructed in the following way: Each node $a_{(i)}$ has an incoming edge from src if $a_{(i)}$ can be the first letter of a word in $\mathcal{L}(\tilde{\alpha})$, and an outgoing edge to snk if it can be the last letter of such a word. Furthermore, for each factor $a_{(i)}b_{(j)}$ that occurs in a word of $\mathcal{L}(\tilde{\alpha})$, there is an edge from $a_{(i)}$ to $b_{(j)}$. As a consequence, there is a one-to-one-correspondence between marked words in $\mathcal{L}(\tilde{\alpha})$ and paths from src to snk in $G_{\tilde{\alpha}}$. To obtain $\mathcal{M}(\alpha)$, we directly interpret $G_{\tilde{\alpha}}$ as NFA over Σ: The source src is the starting state, each node $a_{(i)}$ is a state q_i, and an edge from $a_{(i)}$ to $b_{(j)}$ corresponds to a transition from q_i to q_j when reading b. The sink snk does not become a state; instead, each node with an edge to snk is a final state (hence, $\mathcal{M}(\alpha)$ contains the source state, and one state for every terminal in α). This interpretation allows us to treat occurrence graphs as an alternative notation for a subclass of NFA (namely those where the starting state is not reachable from other states, and for each state q, there is a characteristic terminal a_q such that all transitions to q read a_q). When doing so, we usually omit the occurrence markings on the nodes in graphical representations.

Intuitively, $\mathcal{M}(\alpha)$ treats each terminal of α as a state. Recall that α is not deterministic if there exists words $ua_{(i)}v_1$ and $ua_{(j)}v_2$ in $\mathcal{L}(\tilde{\alpha})$ with $i \neq j$. This corresponds to the situation where, after reading u, $\mathcal{M}(\alpha)$ has to decide between states $a_{(i)}$ and $a_{(j)}$ for the input letter a.

▶ **Example 13.** Let $\alpha := (\varepsilon \vee ((\mathsf{a} \vee \mathsf{b})^+\mathsf{a}))$. Then $\tilde{\alpha} = (\varepsilon \vee ((\mathsf{a}_1 \vee \mathsf{b}_2)^+\mathsf{a}_3))$, and $\mathcal{M}(\alpha)$, the Glushkov automaton of α, is defined as follows:

[1] Most literature, like [9], defines the occurrence graph only implicitly by using sets first, last, and follow, which correspond to the edge from src, the edges to snk, or to the other edges of the graph, respectively. The explicit use of a graph is taken from the k-occurrence automata by Bex et al. [5]. We shall see that an advantage of graphs is that they can be easily extended by describing memory actions to the edges.



To the left, $\mathcal{M}(\alpha)$ is represented as an occurrence graph, to the right in standard NFA notation. Then $\mathcal{M}(\alpha)$ and α are both not deterministic: For $\mathcal{M}(\alpha)$, consider state 1; for α, consider $u = \mathtt{a}_{(1)}$, $v_1 = \mathtt{a}_{(3)}$, $v_2 = \varepsilon$, and the words $u\mathtt{a}_{(1)}v_1$ and $u\mathtt{a}_{(3)}v_2$.

As shown in [9], $\mathcal{L}(\mathsf{DREG}) \subset \mathcal{L}(\mathsf{REG})$ (also see [12, 30], or Lemma 23 below). Like for determinism of regular expressions, the key idea behind our definition of deterministic regex is that a matcher for the expression treats terminals (and variable references) as states. Then an expression is deterministic if the current symbol of the input word always uniquely determines the next state and all necessary variable actions. For regular expressions, non-determinism can only occur when the matcher has to decide between two occurrences of the same terminal symbol; but as regex also need to account for non-determinism that is caused by variable operations or references, their definition of non-determinism is more complicated.

▶ **Definition 14.** An $\alpha \in \mathsf{RX}$ is *not deterministic* if there exist $\rho_1, \rho_2 \in \mathcal{R}(\tilde{\alpha})$ such that any of the following conditions is met for some $r, s_1, s_2 \in (\tilde{\Sigma} \cup \tilde{\Xi} \cup \tilde{\Gamma})^*$ and $\gamma_1, \gamma_2 \in \tilde{\Gamma}^*$:

1. $\rho_1 = r \cdot \gamma_1 \cdot a_{(i)} \cdot s_1$ and $\rho_2 = r \cdot \gamma_2 \cdot a_{(j)} \cdot s_2$ with $a \in \Sigma$ and $i \neq j$,
2. $\rho_1 = r \cdot \gamma_1 \cdot x_{(i)} \cdot s_1$ and $\rho_2 = r \cdot \gamma_2 \cdot \chi_{(j)} \cdot s_2$ with $x \in \Xi$, $\chi \in (\Sigma \cup \Xi)$ and $i \neq j$,
3. $\rho_1 = r \cdot \gamma_1 \cdot \chi_{(i)} \cdot s_1$ and $\rho_2 = r \cdot \gamma_2 \cdot \chi_{(i)} \cdot s_2$ with $\chi \in (\Sigma \cup \Xi)$ and $\gamma_1 \neq \gamma_2$,
4. $\rho_1 = r \cdot \gamma_1$ and $\rho_2 = r \cdot \gamma_2$ with $\gamma_1 \neq \gamma_2$.

Otherwise, α is *deterministic*. We use DRX to denote the set of all deterministic regex, and define $\mathsf{DREG} := \mathsf{DRX} \cap \mathsf{REG}$ as the set of deterministic regular expressions.

▶ **Example 15.** Let $\alpha_1 := (\langle x\colon \mathtt{a}\rangle \vee \mathtt{a})$, $\alpha_2 := (\mathtt{a} \vee \&x)$, $\alpha_3 := (\langle x\colon \varepsilon\rangle \vee \varepsilon)\mathtt{a}$, $\alpha_4 := (\langle x\colon \varepsilon\rangle \vee \varepsilon)$. None of these regex are deterministic, as each α_i meets the i-th condition of Definition 14. We discuss this for α_1: Observe $\tilde{\alpha}_1 = ([_{x(1)}\mathtt{a}_{(2)}]_{x(3)}) \vee \mathtt{a}_{(4)}$. Then choosing $\rho_1 = [_{x(1)}\mathtt{a}_{(2)}]_{x(3)}$ and $\rho_2 = \mathtt{a}_{(4)}$, with $r = \varepsilon$, $\gamma_1 = [_{x(1)}$, $s_1 =]_{x(3)}$, and $\gamma_2 = s_2 = \varepsilon$ shows the condition is met.

Let $\beta_1 := \langle x\colon (\mathtt{a} \vee \mathtt{b})^*\rangle \mathtt{c} \cdot \&x$ and $\beta_2 := \big(\langle x\colon \&y\rangle\langle y\colon \&x \cdot \mathtt{a}\rangle\big)^*$. Both regex are deterministic, with $\mathcal{L}(\beta_1) := \{wcw \mid w \in \{\mathtt{a}, \mathtt{b}\}^*\}$ and $\mathcal{L}(\beta_2) = \{\mathtt{a}^{n^2} \mid n \geq 0\}$ (see Example 2).

Condition 1 of Definition 14 describes cases where non-determinism is caused by two occurrences of the same terminal (γ_1 and γ_2 are included for cases like α_1 in Example 15). If restricted to regular expressions, it is equivalent to the usual definition of deterministic regular expressions. Condition 2 expresses that the matcher has to decide between a variable reference and any other symbol; while in condition 3, the symbol is unique, but there is a non-deterministic choice between variable operations. Finally, condition 4 describes cases where the behaviour of variables is non-deterministic after the end of the word (while one could consider this edge case deterministic, this choice simplifies recursive definitions). In conditions 3 and 4, the definition not only requires that it is clear which variables are reset, but also that it is clear which part of the regex acts on the variables. Hence, $(\langle x\colon \varepsilon\rangle \vee \langle x\colon \varepsilon\rangle)$ is also not deterministic. This is similar to the notion of strong determinism for regular expressions, see [23]. As one might expect, some non-deterministic regexes define DRX-languages:

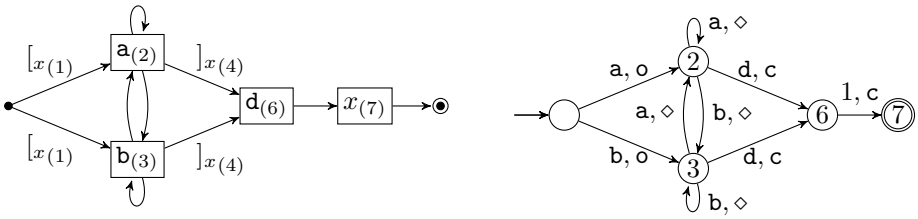
▶ **Example 16.** Let $\Sigma = \{0, 1\}$ and $\alpha := \mathtt{1}^+\langle x\colon \mathtt{0}^*\rangle(\mathtt{1}^+\&x)^*\mathtt{1}^+$. This regex was introduced by Fagin et al. [15], who call its language the "uniform-0-chunk language". Obviously, α is not deterministic (in fact, it satisfies conditions 1, 2, and 3 of Definition 14). Nonetheless, it is possible to express $\mathcal{L}(\alpha)$ with the deterministic regex $\mathtt{1}\big(\mathtt{1}^+ \vee \big(0\langle x\colon 0^*\rangle 1^+(0 \cdot \&x \cdot 1^+)^*\big)\big)$.

We now discuss the conversion from DRX to DTMFA$^{\text{rej}}$, which generalizes the Glushkov construction of $\mathcal{M}(\alpha)$ for regular expressions. The core idea is extending the occurrence graph to a *memory occurrence graph* $G_{\tilde{\alpha}}$, which has two crucial differences: First, instead of only considering terminals, each terminal and each variable reference of a regex α becomes a node. Second, each edge is labelled with a ref-word from $\tilde{\Gamma}^*$ that describes the memory actions (hence, there can be multiple edges from one node to another). In analogy to the occurrence graph, each memory occurrence graph can be directly interpreted as an ε-free TMFA$^{\text{rej}}$.

▶ **Theorem 17.** *Let $\alpha \in$ RX, and let n denote the number of occurrences of terminals and variable references in α. We can construct an $n+2$ state* TMFA$^{\text{rej}}$ *$\mathcal{M}(\alpha)$ with $\mathcal{L}(\mathcal{M}(\alpha)) = \mathcal{L}(\alpha)$ that is deterministic if and only if α is deterministic. In time $O(|\Sigma||\alpha|n)$, the algorithm either*

1. *computes $\mathcal{M}(\alpha)$ if α is deterministic, or*
2. *detects that α is not deterministic.*

▶ **Example 18.** Consider the deterministic regex $\alpha := \langle x\colon (\mathtt{a} \vee \mathtt{b})^+ \rangle \cdot \mathtt{d} \cdot \&x$. Applying the markings yields $\tilde{\alpha} := [_{x(1)}(\mathtt{a}_{(2)} \vee \mathtt{b}_{(3)})^+]_{x(4)} \cdot \mathtt{d}_{(6)} \cdot x_{(7)}$, and $\mathcal{M}(\alpha)$ is the following automaton:



To the left, $\mathcal{M}(\alpha)$ is represented as the memory occurrence graph $G_{\tilde{\alpha}}$, to the right as the DTMFA that can be directly derived from this graph (which uses memory 1 for x).

The construction from the proof of Theorem 17 behaves like the Glushkov construction for regular expressions, with one important difference: On regex that are not deterministic, its running time may be exponential in the number of variables; as there are non-deterministic regex where conversion into a TMFA without ε-transitions requires an exponential amount of transitions. E. g., for $k \geq 1$, let $\alpha := \mathtt{a} \cdot (\varepsilon \vee \langle x_1\colon \varepsilon \rangle) \cdots (\varepsilon \vee \langle x_k\colon \varepsilon \rangle) \cdot \mathtt{b}$ and $\beta := \mathtt{a}\big(\bigvee_{1 \leq i \leq k} \langle x_i\colon \varepsilon \rangle\big)^* \mathtt{b}$. An automaton that is derived with a Glushkov style conversion then contains states q_1 and q_2 that correspond to the terminals; and between these two states, there must be 2^k different transitions to account for all possible combinations of actions on the variables. This suggests that converting a regex into a TMFA without ε-edges is only efficient for deterministic regex; while in general, it is probably advisable to use a construction with ε-edges.

By combining Theorems 17 and 5, due to $n \leq |\alpha|$, we immediately obtain the following:

▶ **Theorem 19.** *Given $\alpha \in$ DRX with n occurrences of terminal symbols or variable references and k variables, and $w \in \Sigma^*$, we can decide in time $O(|\Sigma||\alpha|n + k|w|)$, whether $w \in \mathcal{L}(\alpha)$.*

If we ensure that recalled variables never contain ε (or that only a bounded number of variables references are possible in a row), we can even drop the factor k. For comparison, the membership problem for DREG can be decided in time $O(|\Sigma||\alpha| + |w|)$ when using optimized versions of the Glushkov construction (see [8, 34]), and in $O(|\alpha| + |w| \cdot \log\log|\alpha|)$ with the algorithm by Groz, Maneth, and Staworko [24] that does not compute an automaton.

5 Expressive Power

While Câmpeanu, Salomaa, Yu [10] and Carle and Narendran [11] state pumping lemmas for a class of regex, these do not apply to regex as defined in this paper. However, Lemmas 7 and 10, introduced in Section 3, shall be helpful for proving inexpressibility. A consequence of Lemma 10 is that there are infinite unary $\mathsf{DTMFA}^{\mathsf{rej}}$-languages that are not pumpable (in the sense that certain factors can be repeated arbitrarily often), as this would always lead to an arithmetic progression. It is also possible to demonstrate this phenomenon on larger alphabets, without relying on a trivial modification of the unary case:

▶ **Example 20.** The Fibonacci word F_ω is the infinite word that is the limit of the sequence of words $F_0 := \mathtt{b}$, $F_1 := \mathtt{a}$, and $F_{n+2} := F_{n+1} \cdot F_n$ for all $n \geq 0$. The Fibonacci word has a number of curious properties. In particular, it includes no cubes (i. e., factors www, with $w \neq \varepsilon$). This and various other properties are explained throughout Lothaire [29]. Let

$$\alpha := \mathtt{a}\langle x_0 \colon \mathtt{b}\rangle\langle x_1 \colon \mathtt{a}\rangle\big(\langle x_2 \colon \&x_1\&x_0\rangle\langle x_3 \colon \&x_1\&x_0\&x_1\rangle\langle x_0 \colon \&x_3\&x_2\rangle\langle x_1 \colon \&x_3\&x_2\&x_3\rangle\big)^*.$$

Then $\mathcal{L}(\alpha) = \{F_{4i+3} \mid i \geq 0\}$. Hence, the words of $\mathcal{L}(\alpha)$ converge towards F_ω. The proof of this equivalence is straightforward, but long. It uses that $F_{n+3} = F_{n+1} \cdot F_n \cdot F_{n+1}$ holds for all $n \geq 0$ As F_ω contains no cube, the same applies to all F_n. Thus, $\mathcal{L}(\alpha)$ is a DRX-language that cannot be pumped by repeating factors of sufficiently large words arbitrarily often.

For further separations, we use the following language:

▶ **Example 21.** Let $\alpha := \mathtt{a}^2 \cdot \langle x \colon \mathtt{a}^2\rangle \cdot \big(\langle y \colon \&x \cdot \&x\rangle \cdot \langle x \colon \&y \cdot \&y\rangle\big)^*$. Then $\mathcal{L}(\alpha) = \{\mathtt{a}^{4^i} \mid i \geq 1\}$.

From this, we define an $L \in \mathcal{L}(\mathsf{TMFA})$ with neither $L \in \mathcal{L}(\mathsf{DTMFA}^{\mathsf{rej}})$, nor $L \in \mathcal{L}(\mathsf{DTMFA}^{\mathsf{acc}})$:

▶ **Lemma 22.** *Let* $L := \{\mathtt{a}^{4i+1} \mid i \geq 0\} \cup \{\mathtt{a}^{4^i} \mid i \geq 1\}$. *Then* $L \in \mathcal{L}(\mathsf{TMFA}) \setminus \mathcal{L}(\mathsf{DTMFA})$.

While $\mathsf{DTMFA}^{\mathsf{rej}}$-inexpressibility provides us with a powerful sufficient criterion for DRX-inexpressibility, it is not powerful enough to cover all cases of DRX-inexpressibility. In particular, there are even regular languages that are no DRX-languages:

▶ **Lemma 23.** *Let* $L := \mathcal{L}\big((\mathtt{ab})^*(\mathtt{a} \vee \varepsilon)\big) = \{(\mathtt{ab})^{\frac{1}{2}i} \mid i \geq 0\}$. *Then* $L \in \mathcal{L}(\mathsf{REG}) \setminus \mathcal{L}(\mathsf{DRX})$.

The language L from Lemma 23 is also known to be a non-deterministic regular language (see e. g. [9]). Our proof can be seen as taking the idea behind the characterization of deterministic regular languages from [9], applying it to the specific language L, and also taking variables into account. While this accomplishes the task of proving that deterministic regex share some of the limitations of deterministic regular expressions, the approach does not generalize (at least not in a straightforward manner). In particular, deterministic regex can express regular languages that are not deterministic regular, and are also quite similar to L:

▶ **Example 24.** Let $L := \{(\mathtt{ab})^{\frac{3}{2}i} \mid i \geq 0\}$. Then L is generated by the non-deterministic regular expression $(\mathtt{ababab})^*(\varepsilon \vee (\mathtt{aba}))$, and one can show that L is not a deterministic regular language by using the BKW-algorithm [9] (also [12, 30]) on the minimal DFA M for L. But for $\alpha := \mathtt{a}\langle y \colon \mathtt{b}\rangle\langle x \colon \mathtt{a}\rangle\big(\langle z \colon \&y\rangle\langle y \colon \&x\rangle\langle x \colon \&z\rangle\big)^*$, $\alpha \in \mathsf{DRX}$ and $\mathcal{L}(\alpha) = L$.

The "shifting gadget" that is used in Example 24 can be extended to show a far more general result for unary languages. Considering that $\mathcal{L}(\mathsf{DREG}) \subset \mathcal{L}(\mathsf{REG})$ holds even over unary alphabets (cf. Losemann et al. [28]), the following result might seem surprising:

▶ **Theorem 25.** *For every regular language* L *over a unary alphabet,* $L \in \mathcal{L}(\mathsf{DRX})$.

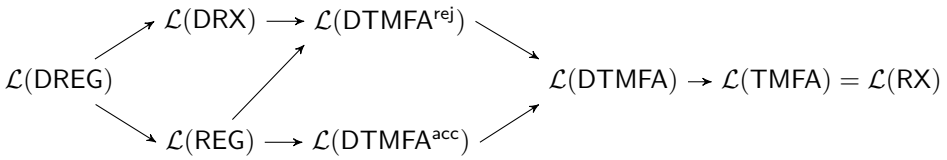


Figure 1 The proper inclusions from Theorem 26. Arrows point from sub- to superset.

As a DFA with n states is converted into a deterministic regex of length $O(n)$, this construction is even efficient. We summarize our observations (also see Figure 1):

▶ **Theorem 26.** $\mathcal{L}(\mathsf{DREG}) \subset \mathcal{L}(\mathsf{DRX}) \subset \mathcal{L}(\mathsf{DTMFA}^{\mathsf{rej}}) \subset \mathcal{L}(\mathsf{DTMFA}) \subset \mathcal{L}(\mathsf{TMFA}) = \mathcal{L}(\mathsf{RX})$.
The following pairs of classes are incomparable: $\mathcal{L}(\mathsf{DRX})$ *and* $\mathcal{L}(\mathsf{REG})$, $\mathcal{L}(\mathsf{DRX})$ *and* $\mathcal{L}(\mathsf{DTMFA}^{\mathsf{acc}})$, *as well as* $\mathcal{L}(\mathsf{DTMFA}^{\mathsf{rej}})$ *and* $\mathcal{L}(\mathsf{DTMFA}^{\mathsf{acc}})$.

We can also use the examples from this section to show that $\mathcal{L}(\mathsf{DRX})$ and $\mathcal{L}(\mathsf{DTMFA}^{\mathsf{rej}})$ are not closed under most of the commonly studied operations on languages:

▶ **Theorem 27.** $\mathcal{L}(\mathsf{DRX})$ *and* $\mathcal{L}(\mathsf{DTMFA}^{\mathsf{rej}})$ *are not closed under the following operations: union, concatenation, reversal, complement, homomorphism, and inverse homomorphism.* $\mathcal{L}(\mathsf{DRX})$ *is also not closed under intersection, and intersection with* DREG*-languages.*

We leave open whether $\mathcal{L}(\mathsf{DTMFA}^{\mathsf{rej}})$ is closed under intersection (with itself or with $\mathcal{L}(\mathsf{DREG})$), but we conjecture that this is not the case. We also leave open whether $\mathcal{L}(\mathsf{DRX})$ and $\mathcal{L}(\mathsf{DTMFA}^{\mathsf{rej}})$ are closed under Kleene plus or star.

6 Two Variants of Determinism

In this section, we examine a restriction and an extension of DRX and DTMFA. We begin with the restriction, which we motivate with the following observation: As shown by Carle and Narendran [11], the intersection problem for regex is undecidable. For DRX, that proof cannot be used, but the result still holds (and by Theorem 17, this extends to DTMFA):

▶ **Theorem 28.** *Given* $\alpha, \beta \in \mathsf{DRX}$, *it is undecidable whether* $\mathcal{L}(\alpha) \cap \mathcal{L}(\beta) = \emptyset$.

As a consequence, DTMFA intersection emptiness problem is also undecidable. Theorem 28 applies even to very restricted DRX, as no variable binding contains a reference to another variable, $|\mathsf{var}(\alpha)| = 2$, and $|\mathsf{var}(\beta)| = 3$. Hence, bounding the number of variables does not make the problem decidable. Instead, the key part seems to be that the variables occur under Kleene stars, which means that they can be reassigned an unbounded amount of times. Following similar observations, Freydenberger and Holldack [20] introduced the following concept: A regex is *variable-star-free (vstar-free)* if each of its plussed sub-regexes contains neither variable references, nor variable bindings. Analogously, we call a TMFA *memory-cycle-free* if it contains no cycle with a *memory transition* (a transition in a TMFA that is a memory recall, or that contains memory actions other than $\diamond$). Let $\mathsf{RX}_{\mathsf{vsf}}$ be the set of all vstar-free regex, and $\mathsf{DRX}_{\mathsf{vsf}} = \mathsf{RX}_{\mathsf{vsf}} \cap \mathsf{DRX}$. Let $\mathsf{TMFA}_{\mathsf{mcf}}$ be the set of all memory-cycle-free TMFA, and define $\mathsf{DTMFA}_{\mathsf{mcf}}$, $\mathsf{TMFA}^{\mathsf{rej}}_{\mathsf{mcf}}$,... analogously. The proof of Theorem 17 allows us to conclude that $\mathcal{M}(\alpha) \in \mathsf{DTMFA}_{\mathsf{mcf}}$ holds for every $\alpha \in \mathsf{DRX}_{\mathsf{vsf}}$. Likewise, we can use the proof of Theorem 4 to conclude $\mathcal{L}(\mathsf{TMFA}_{\mathsf{mcf}}) = \mathcal{L}(\mathsf{RX}_{\mathsf{vsf}})$. Note that for ε-free $\mathsf{DTMFA}_{\mathsf{mcf}}$, the membership problem can be decided in time $O(|Q| + |w|)$, as the preprocessing step of Theorem 5 is not necessary (as only a bounded number of variable references is possible in each run). Likewise, we can drop the factor k from Theorem 19 when restricted to $\mathsf{DRX}_{\mathsf{vsf}}$.

As shown by Freydenberger [19], it is decidable in PSPACE whether $\bigcap_{i=1}^{n} \mathcal{L}(\alpha_i) = \emptyset$ for $\alpha_1, \ldots, \alpha_n \in \mathsf{RX}_{\mathsf{vsf}}$. By combining the proof for this with some ideas from another construction from [19], we encode the intersection emptiness problem for $\mathsf{TMFA}_{\mathsf{mcf}}$ in the *existential theory of concatenation with regular constraints* (a PSPACE-decidable, positive logic on words, see Diekert [13], Diekert, Jeż, Plandowski [14]). This yields the following:

▶ **Theorem 29.** *Given $M_1, \ldots, M_n \in \mathsf{TMFA}_{\mathsf{mcf}}$, we can decide whether $\bigcap_{i=1}^{n} \mathcal{L}(M_i) = \emptyset$ in* PSPACE*. The problem is* PSPACE*-hard, even if restricted to $\mathcal{L}(\alpha) \cap \mathcal{L}(\beta)$, $\alpha \in \mathsf{DRX}_{\mathsf{vsf}}$ and $\beta \in$ DREG (if the size of Σ is not bounded), or to $\mathcal{L}(\alpha) \cap \mathcal{L}(M)$, $\alpha \in \mathsf{DRX}_{\mathsf{vsf}}$ and $M \in$ DFA.*

The unbounded size of Σ comes from the PSPACE-hardness of the intersection emptiness problem for DRX by Martens et al. [31], which has the same requirement. Using the existential theory of concatenation for the upper bound might seem conceptually excessive – but this cannot be avoided (see Section A.18 in the full version of the paper [21]).

We now combine the proofs of Theorems 6 and 29, and observe:

▶ **Theorem 30.** *Given $M_1, M_2 \in \mathsf{DTMFA}_{\mathsf{mcf}}$, $\mathcal{L}(M_1) \subseteq \mathcal{L}(M_2)$ can be decided in* PSPACE*.*

Obviously, this implies that equivalence for $\mathsf{DTMFA}_{\mathsf{mcf}}$ is decidable in PSPACE, and, furthermore, this also holds for $\mathsf{DRX}_{\mathsf{vsf}}$, which is an interesting contrast to non-deterministic $\mathsf{RX}_{\mathsf{vsf}}$: As shown by Freydenberger [18], equivalence (and, hence, inclusion and minimization) are undecidable for $\mathsf{RX}_{\mathsf{vsf}}$ (while [18] does not explicitly mention the concept, the regex in that proof are vstar-free, as discussed in [20]). Hence, Theorem 30 also yields a minimization algorithm for $\mathsf{DRX}_{\mathsf{vsf}}$ and $\mathsf{DTMFA}_{\mathsf{mcf}}$ that works in PSPACE (enumerate all smaller candidates and check equivalence). We leave open whether this is optimal, but observe that even for DREG, minimization is NP-complete, see Niewerth [33].

Next, we discuss a potential extension of determinism. One could argue that Definition 14 is overly restrictive; e. g., consider $\alpha := \langle x \colon \mathtt{a}^+ \rangle \langle y \colon \mathtt{b}+ \rangle \mathtt{c}(\&x \vee \&y)$. Then α is not deterministic; but as the contents of x and y always start with a or b (respectively), deterministic choices between $\&x$ and $\&y$ are possible by looking at the current letter of the input word. Analogous observations can be made for TMFA. More precisely, we define the notion of *ℓ-deterministic* TMFA as a relaxation of the criteria of DTMFA: In contrast to the latter, an ℓ-deterministic TMFA can have states q with multiple memory recall-transitions, as long as these recall distinct memories, and if q is reached in some computation, then for each pair of these recalled memories, the contents differ in the first ℓ positions. First, note that this does not increase the expressive power (intuitively, storing the length ℓ prefixes of the memory contents allows making ℓ-deterministic memory recall transitions deterministic):

▶ **Proposition 31.** *Let $\ell \geq 1$. For every ℓ-deterministic $M \in$ DTMFA, there is an $M' \in$ DTMFA with $\mathcal{L}(M) = \mathcal{L}(M')$.*

For the sake of the argument, let $\alpha \in \mathsf{RX}$ be ℓ-deterministic if and only if $\mathcal{M}(\alpha)$ is.

▶ **Proposition 32.** *For every $\ell \geq 1$, deciding whether a* TMFA *is ℓ-deterministic is* PSPACE*-complete. The problem is* coNP*-complete if the input is restricted to $\mathsf{TMFA}_{\mathsf{mcf}}$. These lower bounds hold even if we restrict the input to* RX *and $\mathsf{RX}_{\mathsf{vsf}}$, respectively.*

Hence, while we can decide efficiently whether a TMFA or a regex is deterministic, detecting ℓ-determinism is costly, even for $\ell = 1$. The same holds if we adapt the definition to distinguish between variables and terminals (see Section A.21 in the full version of the paper [21]).

7 Conclusions and Further Directions

Based on TMFA, an automaton model for regex, we extended the notion of determinism from regular expressions to regex. Although the resulting language class cannot express all regular languages, it is still rich; and by using a generalization of the Glushkov construction, deterministic regex can be converted into a DTMFA, and the membership problem can then be solved quite efficiently. Although we did not discuss this, the construction is also compatible with the Glushkov construction with counters by Gelade, Gyssens, Martens [23]. Hence, one can add counters to DRX and DTMFA without affecting the complexity of membership.

Many challenging questions remain open, for example: Can the more advanced results for DREG be adapted to DRX, i. e., can $\mathcal{M}(\alpha)$ be computed more efficiently (as in [8, 34]), or is it even possible, like in [24], to avoid computing $\mathcal{M}(\alpha)$? Is effective minimization possible for DTMFA or DRX? Is it decidable whether a DTMFA defines a DRX-language? Are inclusion and equivalence decidable for DRX or DTMFA? Can determinism be generalized to larger classes of regex without making the membership problem intractable?

Acknowledgements. The authors thank Wim Martens for helpful feedback, Matthias Niewerth, for pointing out that v_n must be a factor of p_n in the jumping lemma, and Martin Braun, for creating a library and tool for DRX and DTMFA (available at [6]).

References

1 Abigail. Re: Random number in perl. Posting in the newsgroup comp.lang.perl.misc, October 1997. Message-ID slrn64sudh.qp.abigail@betelgeuse.wayne.fnx.com.

2 Alfred V. Aho. Algorithms for finding patterns in strings. In Jan van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume A, chapter 5, pages 255–300. Elsevier, Amsterdam, 1990.

3 Dana Angluin. Finding patterns common to a set of strings. *J. Comput. Syst. Sci.*, 21:46–62, 1980.

4 Pablo Barceló, Carlos A. Hurtado, Leonid Libkin, and Peter T. Wood. Expressive languages for path queries over graph-structured data. In *Proc. PODS 2010*, 2010.

5 Geert Jan Bex, Wouter Gelade, Frank Neven, and Stijn Vansummeren. Learning deterministic regular expressions for the inference of schemas from XML data. *ACM Trans. Web*, 4(4):14, 2010.

6 Martin Braun. moar – Deterministic Regular Expressions with Backreferences, 2016. Accessed December 2016. URL: `https://github.com/s4ke/moar`.

7 Tim Bray, Jean Paoli, C. M. Sperberg-McQueen, Eve Maler, and François Yergeau. Extensible markup language XML 1.0 (fifth edition). W3C recommendation. Technical Report `https://www.w3.org/TR/2008/REC-xml-20081126/`, W3C, November 2008.

8 Anne Brüggemann-Klein. Regular expressions into finite automata. *Theor. Comput. Sci.*, 120(2):197–213, 1993.

9 Anne Brüggemann-Klein and Derick Wood. One-unambiguous regular languages. *Inf. Comput.*, 142(2):182–206, 1998.

10 Cezar Câmpeanu, Kai Salomaa, and Sheng Yu. A formal study of practical regular expressions. *Int. J. Found. Comput. Sci.*, 14:1007–1018, 2003.

11 Benjamin Carle and Paliath Narendran. On extended regular expressions. In *Proc. LATA 2009*, 2009.

12 Wojciech Czerwinski, Claire David, Katja Losemann, and Wim Martens. Deciding definability by deterministic regular expressions. In *Proc. FOSSACS 2013*, pages 289–304, 2013.

13 Volker Diekert. Makanin's Algorithm. In *Algebraic Combinatorics on Words* [29], chapter 12.

14 Volker Diekert, Artur Jeż, and Wojciech Plandowski. Finding all solutions of equations in free groups and monoids with involution. In *Proc. CSR 2014*, 2014.

15 Ronald Fagin, Benny Kimelfeld, Frederick Reiss, and Stijn Vansummeren. Document spanners: A formal approach to information extraction. *J. ACM*, 62(2):12, 2015.

16 Henning Fernau and Markus L. Schmid. Pattern matching with variables: A multivariate complexity analysis. *Inform. Comput.*, 242:287–305, 2015.

17 Henning Fernau, Markus L. Schmid, and Yngve Villanger. On the parameterised complexity of string morphism problems. *Theory Comput. Syst.*, 59(1):24–51, 2016.

18 Dominik D. Freydenberger. Extended regular expressions: Succinctness and decidability. *Theory Comput. Sys.*, 53(2):159–193, 2013.

19 Dominik D. Freydenberger. A logic for document spanners. In *Proc. ICDT 2017*, 2017. Accepted. Available at `http://ddfy.de/publications/F-ALfDS.html`.

20 Dominik D. Freydenberger and Mario Holldack. Document spanners: From expressive power to decision problems. In *Proc. ICDT 2016*, 2016.

21 Dominik D. Freydenberger and Markus L. Schmid. Deterministic regular expressions with back-references. A version of this paper that also includes the Appendix. URL: `http://ddfy.de/publications/FS-DREwBR.html`.

22 Shudi (Sandy) Gao, C. M. Sperberg-McQueen, and Henry S. Thompson. W3C XML schema definition language (XSD) 1.1 part 1: Structures. Technical Report `https://www.w3.org/TR/2012/REC-xmlschema11-1-20120405/`, W3C, April 2012.

23 Wouter Gelade, Marc Gyssens, and Wim Martens. Regular expressions with counting: Weak versus strong determinism. *SIAM J. Comput.*, 41(1):160–190, 2012.

24 Benoît Groz, Sebastian Maneth, and Slawek Staworko. Deterministic regular expressions in linear time. In *Proc. PODS 2012*, 2012.

25 John E. Hopcroft and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, 1979.

26 S. C. Kleene. Representation of events in nerve nets and finite automata. In C. E. Shannon, J. McCarthy, and W. R. Ashby, editors, *Automata Studies*, pages 3–42. Princeton University Press, Princeton, NJ, 1956.

27 Markus Latte and Matthias Niewerth. Definability by weakly deterministic regular expressions with counters is decidable. In *Proc. MFCS 2015*, 2015.

28 Katja Losemann, Wim Martens, and Matthias Niewerth. Closure properties and descriptional complexity of deterministic regular expressions. *Theor. Comput. Sci.*, 627:54–70, 2016.

29 M. Lothaire. *Algebraic Combinatorics on Words*, volume 90 of *Encyclopedia of mathematics and its applications*. Cambridge University Press, 2002.

30 Ping Lu, Joachim Bremer, and Haiming Chen. Deciding determinism of regular languages. *Theory Comput. Syst.*, 57(1):97–139, 2015.

31 Wim Martens, Frank Neven, and Thomas Schwentick. Complexity of decision problems for XML schemas and chain regular expressions. *SIAM J. Comput.*, 39(4):1486–1530, 2009.

32 Makoto Murata, Dongwon Lee, Murali Mani, and Kohsuke Kawaguchi. Taxonomy of XML schema languages using formal language theory. *ACM TOIT*, 5(4):660–704, 2005.

33 Matthias Niewerth. *Data Definition Languages for XML Repository Management Systems*. PhD thesis, TU Dortmund, 2015. URL: `http://www.theoinf.uni-bayreuth.de/en/downloads/PHD_Niewerth.pdf`.

34 Jean-Luc Ponty, Djelloul Ziadi, and Jean-Marc Champarnaud. A new quadratic algorithm to convert a regular expression into an automaton. In *Proc. WIA'96*, 1996.

35 Markus L. Schmid. Characterising REGEX languages by regular languages equipped with factor-referencing. *Inform. Comput.*, 249:1–17, 2016.