Regular Expressions with Counting: Weak versus Strong Determinism

Wouter Gelade[1⋆], Marc Gyssens[1], and Wim Martens[2⋆⋆]

[1] Hasselt University and Transnational University of Limburg
School for Information Technology
{firstname.lastname}@uhasselt.be
[2] Technical University of Dortmund
{firstname.lastname}@udo.edu

Abstract. We study deterministic regular expressions extended with the counting operator. There exist two notions of determinism, strong and weak determinism, which almost coincide for standard regular expressions. This, however, changes dramatically in the presence of counting. In particular, we show that weakly deterministic expressions with counting are exponentially more succinct and strictly more expressive than strongly deterministic ones, even though they still do not capture all regular languages. In addition, we present a finite automaton model with counters, study its properties and investigate the natural extension of the Glushkov construction translating expressions with counting into such counting automata. This translation yields a deterministic automaton if and only if the expression is strongly deterministic. These results then also allow to derive upper bounds for decision problems for strongly deterministic expressions with counting.

1 Introduction

The use of regular expressions (REs) is quite widespread and includes applications in bioinformatics [17], programming languages [23], model checking [22], XML schema languages [21], etc. In many cases, the standard operators are extended with additional ones to facilitate usability. A popular such operator is the counting operator allowing for expressions of the form "$a^{2,4}$", defining strings containing at least two and at most four a's, which is used for instance in Egrep [9] and Perl [23] patterns and in the XML schema language XML Schema [21].

In addition to expanding the vocabulary of REs, subclasses of REs have been investigated to alleviate, e.g., the matching problem. For instance, in the context of XML and SGML, the strict subclasses of weakly and strongly deterministic regular expressions have been introduced. Weak determinism (also called one-unambiguity [2]) intuitively requires that, when matching a string from left to

⋆ Research Assistant of the Fund for Scientific Research – Flanders (Belgium).
⋆⋆ Supported by the North-Rhine Westphalian Academy of Sciences, Humanities and Arts; and the Stiftung Mercator Essen.

right against an expression, it is always clear against which position in the expression the next symbol must be matched. For example, the expression $(a+b)^*a$ is not weakly deterministic, but the equivalent expression $b^*a(b^*a)^*$ is. Strong determinism intuitively requires additionally that it is also clear *how* to go from one position to the next. For example, $(a^*)^*$ is weakly deterministic, but not strongly deterministic since it is not clear over which star one should iterate when going from one a to the next.

While weak and strong determinism coincide for standard regular expressions [1][3], this situation changes completely when counting is involved. Firstly, the algorithm for deciding whether an expression is weakly deterministic is non-trivial [13]. For instance, $(b?a^{2,3})^{2,2}b$ is weakly deterministic, but the very similar $(b?a^{2,3})^{3,3}b$ is not. So, the amount of non-determinism introduced depends on the concrete values of the counters. Second, as we will show, weakly deterministic expressions with counting are strictly more expressive than strongly deterministic ones. Therefore, the aim of this paper is an in-depth study of the notions of weak and strong determinism in the presence of counting w.r.t. expressiveness, succinctness, and complexity. In particular, our contributions are the following:

– We give a complete overview of the expressive power of the different classes of deterministic expressions with counting. We show that strongly deterministic expressions with counting are equally expressive as standard deterministic expressions. Weakly deterministic expressions with counting, on the other hand, are more expressive than strongly deterministic ones, except for unary languages, on which they coincide. However, not all unary regular languages are definable by weakly deterministic expressions with counting (Section 3).
– We investigate the difference in succinctness between strongly and weakly deterministic expressions with counting, and show that weakly deterministic expressions can be exponentially more succinct than strongly deterministic ones. This result prohibits an efficient algorithm translating a weakly deterministic expression into an equivalent strongly deterministic one, if such an expression exists. This contrasts with the situation of standard expressions where such a linear time algorithm exists [1] (Section 4).
– We present an automaton model extended with counters, counter NFAs (CNFAs), and investigate the complexity of some related problems. For instance, it is shown that boolean operations can be applied efficiently to CDFAs, the deterministic counterpart of CNFAs (Section 5).
– Bruggemann-Klein [1] has shown that the Glushkov construction, translating regular expressions into NFAs, yields a DFA if and only if the original expression is deterministic. We investigate the natural extension of the Glushkov construction to expressions with counters, converting expressions to CNFAs. We show that the resulting automaton is deterministic if and only if the original expression is strongly deterministic (Section 6).

[3] Brüggemann-Klein [1] did not study strong determinism explicitly, although she did study strong unambiguity. However, she gives a procedure to transform expressions into *star normal form* which rewrites weakly deterministic expressions into equivalent strongly deterministic ones in linear time.

– Combining the results of Section 5, concerning CDFAs, with the latter result then also allows to infer better upper bounds on the inclusion and equivalence problem of strongly deterministic expressions with counting. Further, we show that testing whether an expression with counting is strongly deterministic can be done in cubic time, as is the case for weak determinism [13] (Section 7).

The original motivation for this work comes from the XML schema language XML Schema, which uses weakly deterministic expressions with counting. However, it is also noted by Sperberg-McQueen [20], one of its developers, that *"Given the complications which arise from [weakly deterministic expressions], it might be desirable to also require that they be strongly deterministic as well [in XML Schema]."* The design decision for weak determinism is probably inspired by the fact that it is the natural extension of the notion of determinism for standard expressions, and a lack of a detailed analysis of their differences when counting is allowed. A detailed examination of strong and weak determinism of regular expressions with counting intends to fill this gap.

Related work: Apart from the work already mentioned, there are several automata based models for different classes of expressions with counting with as main application XML Schema validation, by Kilpelainen and Tuhkanen [12], Zilio and Lugiez [4], and Sperberg-McQueen [20]. Here, Sperberg-McQueen introduces the extension of the Glushkov construction which we study in Section 6. We introduce a new automata model in Section 5 as none of these models allow to derive all results in Sections 5 and 6. Further, Sperberg-McQueen [20] and Koch and Scherzinger [14] introduce a (slightly different) notion of strongly deterministic expression with and without counting, respectively. We follow the semantic meaning of Sperberg-McQueen's definition, while using the technical approach of Koch and Scherzinger. Finally, Kilpelainen [10] shows that inclusion for weakly deterministic expressions with counting is coNP-hard; and Colazzo, Ghelli, and Sartiani [3] have investigated the inclusion problem involving subclasses of deterministic expressions with counting. Seidl et al. also investigate counting constraints in XML schema languages by adding Presburger constraints to regular languages [18]. Concerning deterministic languages without counting, the seminal paper is by Bruggemann-Klein and Wood [2] where, in particular, it is shown to be decidable whether a language is definable by a deterministic regular expression. Conversely, general regular expressions with counting have also received quite some attention [7, 8, 11, 16].

2 Preliminaries

Let $\mathbb{N}$ denote the natural numbers $\{0, 1, 2, \ldots\}$. For the rest of the paper, Σ always denotes a finite alphabet. The set of regular expressions over Σ, denoted by $\mathrm{RE}(\Sigma)$, is defined as follows: ε and every Σ-symbol is in $\mathrm{RE}(\Sigma)$; and whenever r and s are in $\mathrm{RE}(\Sigma)$, then so are (rs), $(r + s)$, and $(s)^*$. For readability, we usually omit parentheses in examples. The language defined by a regular expression r, denoted by $L(r)$, is defined as usual. By $\mathrm{RE}(\Sigma,\#)$ we denote $\mathrm{RE}(\Sigma)$

extended with *numerical occurrence constraints* or *counting*. That is, when r is an RE(Σ,#)-expression then so is $r^{k,\ell}$ for $k \in \mathbb{N}$ and $\ell \in \mathbb{N}_0 \cup \{\infty\}$ with $k \leq \ell$. Here, $\mathbb{N}_0$ denotes $\mathbb{N} \setminus \{0\}$. Furthermore, $L(r^{k,\ell}) = \bigcup_{i=k}^{\ell} L(r)^i$. We use $r?$ to abbreviate $(r+\varepsilon)$. Notice that r^* is simply an abbreviation for $r^{0,\infty}$. Therefore, we do not consider the $*$-operator in the context of RE(Σ,#). The *size* of a regular expression r in RE(Σ,#), denoted by $|r|$, is the number of Σ-symbols and operators occurring in r plus the sizes of the binary representations of the integers. An RE(Σ,#) expression r is *nullable* if $\varepsilon \in L(r)$. We say that an RE(Σ,#) r is in *normal form* if for every nullable subexpression $s^{k,l}$ of r we have $k = 0$. Any RE(Σ,#) can easily be normalized in linear time. Therefore, we assume that all expressions used in this paper are in normal form. Sometimes we will use the following observation, which follows directly from the definitions:

Remark 1. A subexpression $r^{k,\ell}$ is nullable if and only if $k = 0$.

Weak determinism. For an RE(Σ,#) r, let Char(r) be the set of Σ-symbols occurring in r. A *marked regular expression with counting over Σ* is a regular expression over $\Sigma \times \mathbb{N}$ in which every ($\Sigma \times \mathbb{N}$)-symbol occurs at most once. We denote the set of all these expressions by MRE(Σ,#). Formally, $\overline{r} \in$ MRE(Σ,#) if $\overline{r} \in$ RE($\Sigma \times \mathbb{N}$,#) and, for every subexpression $\overline{s}\,\overline{s}'$ or $\overline{s} + \overline{s}'$ of $\overline{r}$, Char($\overline{s}$) $\cap$ Char($\overline{s}'$) $= \emptyset$. A *marked string* is a string over $\Sigma \times \mathbb{N}$ (in which ($\Sigma \times \mathbb{N}$)-symbols can occur more than once). When $\overline{r}$ is a marked regular expression, $L(\overline{r})$ is therefore a set of marked strings.

The demarking of a marked expression is obtained by deleting these integers. Formally, the demarking of $\overline{r}$ is dm($\overline{r}$), where dm : MRE(Σ,#) $\to$ RE(Σ,#) is defined as dm(ε) := ε, dm((a,i)) := a, dm($\overline{rs}$) := dm($\overline{r}$)dm($\overline{s}$), dm($\overline{r+s}$) := dm($\overline{r}$) + dm($\overline{s}$), and dm($\overline{r^{k,\ell}}$) := dm($\overline{r}$)$^{k,\ell}$. Any function m : RE(Σ,#) $\to$ MRE(Σ,#) such that for every $r \in$ RE(Σ,#) it holds that dm(m(r)) $= r$ is a valid *marking* function. For conciseness and readability, we will from now on write a_i instead of (a,i) in marked regular expressions. For instance, a *marking* of $(a+b)^{1,2}a+bc$ is $(a_1+b_1)^{1,2}a_2+b_2c_1$. The markings and demarkings of strings are defined analogously. For the rest of the paper, we usually leave the actual marking function m implicit and denote by $\overline{r}$ a marking of the expression r. Likewise $\overline{w}$ will denote a marking of a string w. We always use overlined letters to denote marked expressions, symbols, and strings.

Definition 2. An RE(Σ,#) expression r is *weakly deterministic* (also called *one-unambiguous*) if, for all strings $\overline{u},\overline{v},\overline{w} \in$ Char($\overline{r}$)* and all symbols $\overline{a},\overline{b} \in$ Char($\overline{r}$), the conditions $\overline{uav}, \overline{ubw} \in L(\overline{r})$ and $\overline{a} \neq \overline{b}$ imply that $a \neq b$.

A regular language is *weakly deterministic with counting* if it is defined by some weakly deterministic RE(Σ,#) expression. The classes of all weakly deterministic languages with counting, respectively, without counting, are denoted by DET$_W^{\#}(\Sigma)$, respectively, DET$_W(\Sigma)$.

Intuitively, an expression is weakly deterministic if, when matching a string against the expression from left to right, we always know against which symbol in the expression we must match the next symbol, without looking ahead in the

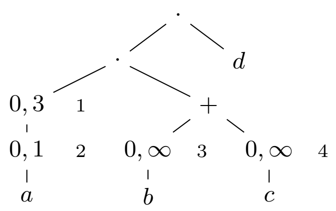

Fig. 1. Parse tree of $(a^{0,1})^{0,3}(b^{0,\infty} + c^{0,\infty})d$. Counter nodes are numbered from 1 to 4.

string. For instance, $(a + b)^*a$ and $(a^{2,3} + b)^{3,3}b$ are not weakly deterministic, while $b^*a(b^*a)^*$ and $(a^{2,3} + b)^{2,2}b$ are.

Strong determinism. Intuitively, an expression is weakly deterministic if, when matching a string from left to right, we always know *where* we are in the expression. For a strongly deterministic expression, we will additionally require that we always know *how* to go from one position to the next. Thereto, we distinguish between going *forward* in an expression and *backward* by *iterating* over a counter. For instance, in the expression $(ab)^{1,2}$ going from a to b implies going forward, whereas going from b to a iterates backward over the counter.

Therefore, an expression such as $((a+\varepsilon)(b+\varepsilon))^{0,2}$ will not be strongly deterministic, although it is weakly deterministic. Indeed, when matching ab, we can go from a to b by either going forward or by iterating over the counter. By the same token, also $(a^{1,2})^{3,4}$ is not strongly deterministic, as we have a choice of counters over which to iterate when reading multiple a's. Conversely, $(a^{2,2})^{3,4}$ is strongly deterministic as it is always clear over which counter we must iterate.

For the definition of strong determinism, we follow the semantic meaning of the definition by Sperberg-McQueen [20], while using the formal approach of Koch and Scherzinger [14] (who called the notion *strong one-unambiguity*)[4]. We denote the *parse tree* of an RE(Σ,#) expression r by pt(r). Figure 1 contains the parse tree of the expression $(a^{0,1})^{0,3}(b^{0,\infty} + c^{0,\infty})d$.

A *bracketing of a regular expression* r is a labeling of the counter nodes of pt(r) by distinct indices. Concretely, we simply number the nodes according to the depth-first left-to-right ordering. The bracketing $\widetilde{r}$ of r is then obtained by replacing each subexpression $s^{k,\ell}$ of r with index i with $([_i s]_i)^{k,\ell}$. Therefore, a bracketed regular expression is a regular expression over alphabet $\Sigma \uplus \Gamma$, where $\Gamma := \{[_i,]_i \mid i \in \mathbb{N}\}$. For example, $([_1([_2a]_2)^{0,1}]_1)^{0,3}(([_3b]_3)^{0,\infty} + ([_4c]_4)^{0,\infty})d$ is a bracketing of $(a^{0,1})^{0,3}(b^{0,\infty}+c^{0,\infty})d$, for which the parse tree is shown in Figure 1. We say that a string w in $\Sigma \uplus \Gamma$ is *correctly bracketed* if w has no substring of the form $[_i]_i$. That is, we do not allow a derivation of ε in the derivation tree.

Definition 3. A regular expression r is strongly deterministic with counting if r is weakly deterministic and there do not exist strings u, v, w over $\Sigma \cup \Gamma$, strings

[4] The difference with Koch and Scherzinger is that we allow different derivations of ε while they forbid this. For instance, $a^* + b^*$ is strongly deterministic in our definition, but not in theirs, as ε can be matched by both a^* and b^*.

$\alpha \neq \beta$ over Γ, and a symbol $a \in \Sigma$ such that $u\alpha av$ and $u\beta aw$ are both correctly bracketed and in $L(\widetilde{r})$.

A standard regular expression (without counting) is strongly deterministic if the expression obtained by replacing each subexpression of the form r^* with $r^{0,\infty}$ is strongly deterministic with counting. The class $\mathrm{DET}^{\#}_{S}(\Sigma)$, respectively, $\mathrm{DET}_{S}(\Sigma)$, denotes all languages definable by a strongly deterministic expressions with, respectively, without, counting.

3 Expressive power

Brüggemann-Klein and Wood [2] proved that for any alphabet Σ $\mathrm{DET}_W(\Sigma)$ forms a strict subclass of the regular languages, denoted $\mathrm{REG}(\Sigma)$. The complete picture of the relative expressive power depends on the size of Σ, as shown by the following theorem.

Theorem 4. *For every alphabet Σ,*

$$DET_S(\Sigma) = DET_W(\Sigma) = DET^{\#}_{S}(\Sigma) = DET^{\#}_{W}(\Sigma) \subsetneq REG(\Sigma) \quad (\textit{if } |\Sigma| = 1)$$

$$DET_S(\Sigma) = DET_W(\Sigma) = DET^{\#}_{S}(\Sigma) \subsetneq DET^{\#}_{W}(\Sigma) \subsetneq REG(\Sigma) \quad (\textit{if } |\Sigma| \geq 2)$$

Proof. The equality $\mathrm{DET}_S(\Sigma) = \mathrm{DET}_W(\Sigma)$ is already implicit in the work of Brüggemann-Klein [1]. By this result and by definition, all inclusions from left to right already hold. It therefore suffices to show that (1) $\mathrm{DET}^{\#}_{S}(\Sigma) \subseteq \mathrm{DET}_S(\Sigma)$ for arbitrary alphabets, (2) $\mathrm{DET}^{\#}_{W}(\Sigma) \subseteq \mathrm{DET}^{\#}_{S}(\Sigma)$ for unary alphabets, (3) $\mathrm{DET}^{\#}_{S}(\Sigma) \subsetneq \mathrm{DET}^{\#}_{W}(\Sigma)$ for binary alphabets, and (4) $\mathrm{DET}^{\#}_{W}(\Sigma) \subsetneq \mathrm{REG}(\Sigma)$ for unary alphabets.
(1): We show that each strongly deterministic expression with counting can be transformed into a strongly deterministic expression without counting. This is quite non-trivial, but the crux is to unfold each counting operator in a smart manner, taking special care of nullable expressions.
(2): The crux of this proof lies in Lemma 5. It is well known and easy to see that the minimal DFA for a regular language over a unary alphabet is defined either by a simple chain of states (sometimes also called a *tail* [19]), or a chain followed by a cycle.The languages in $\mathrm{DET}^{\#}_{W}(\Sigma)$ can be defined in this manner. The following lemma adds to that, that for weakly deterministic regular expressions, only one node in this cycle can be final. The theorem then follows as any such language can be defined by a strongly deterministic expression.

Lemma 5. *Let $\Sigma = \{a\}$, and $L \in REG(\Sigma)$, then $L \in DET^{\#}_{W}(\Sigma)$ if and only if L is definable by a DFA which is either a chain, or a chain followed by a cycle, for which at most one of the cycle nodes is final.*

(3 and 4): Witnesses for non-inclusion are the languages defined by $(a^{2,3}b?)^*$ and $(aaa)^*(a + aa)$, respectively. Both languages can be shown not to be in $\mathrm{DET}_W(\Sigma)$ [2]. The theorem then follows from the above results.

4 Succinctness

In Section 3 we learned that $\text{DET}^{\#}_{W}(\Sigma)$ strictly contains $\text{DET}^{\#}_{S}(\Sigma)$, prohibiting a translation from weak to strong deterministic expressions with counting. However, one could still hope for an efficient algorithm which, given a weakly deterministic expression known to be equivalent to a strong deterministic one, constructs this expression. However, this is not the case:

Theorem 6. *For every $n \in \mathbb{N}$, there exists an $r \in RE(\Sigma,\#)$ over alphabet $\{a\}$ which is weakly deterministic and of size $\mathcal{O}(n)$ such that every strongly deterministic expression s, with $L(r) = L(s)$, is of size at least 2^n.*

The above theorem holds for the family of languages defined by $(a^{2^n+1,2^{n+1}})^{1,2}$, each of which is weakly deterministic and defines all strings with a's of length from $2^n + 1$ to 2^{n+2}, except for the string $a^{2^{n+1}+1}$. These expressions, in fact, where introduced by Kilpelainen when studying the inclusion problem for weakly deterministic expressions with counting [10].

5 Counter automata

Let C be a set of *counter variables* and $\alpha : C \to \mathbb{N}$ be a function assigning a value to each counter variable. We inductively define *guards* over C, denoted $\text{Guard}(C)$, as follows: for every $\text{cv} \in C$ and $k \in \mathbb{N}$, we have that true, false, $\text{cv} = k$, and $\text{cv} < k$ are in $\text{Guard}(C)$. Moreover, when $\phi_1, \phi_2 \in \text{Guard}(C)$, then so are $\phi_1 \wedge \phi_2$, $\phi_1 \vee \phi_2$, and $\neg\phi_1$. For $\phi \in \text{Guard}(C)$, we denote by $\alpha \models \phi$ that α models ϕ, i.e., that applying the value assignment α to the counter variables results in satisfaction of ϕ.

An *update* is a set of statements of the form cv++ and reset(cv) in which every $\text{cv} \in C$ occurs at most once. By $\text{Update}(C)$ we denote the set of all updates.

Definition 7. A non-deterministic *counter automaton* (CNFA) is a 6-tuple $A = (Q, q_0, C, \delta, F, \tau)$ where Q is the finite set of states; $q_0 \in Q$ is the initial state; C is the finite set of counter variables; $\delta : Q \times \Sigma \times \text{Guard}(C) \times \text{Update}(C) \times Q$ is the transition relation; $F : Q \to \text{Guard}(C)$ is the acceptance function; and $\tau : C \to \mathbb{N}$ assigns a maximum value to every counter variable.

Intuitively, A can make a transition (q, a, ϕ, π, q') whenever it is in state q, reads a, and guard ϕ is true under the current values of the counter variables. It then updates the counter variables according to the update π, in a way we explain next, and moves into state q'. To explain the update mechanism formally, we introduce the notion of configuration. Thereto, let $\max(A) = \max\{\tau(c) \mid c \in C\}$. A *configuration* is a pair (q, α) where $q \in Q$ is the current state and $\alpha : C \to \{1, \ldots, \max(A)\}$ is the function mapping counter variables to their current value. Finally, an update π transforms α into $\pi(\alpha)$ by setting $\text{cv} := 1$, when $\text{reset(cv)} \in \pi$, and $\text{cv} := \text{cv} + 1$ when $\text{cv++} \in \pi$ and $\alpha(\text{cv}) < \tau(\text{cv})$. Otherwise, the value of cv remains unaltered.

Let α_0 be the function mapping every counter variable to 1. The *initial configuration* γ_0 is (q_0, α_0). A configuration (q, α) is *final* if $\alpha \models F(q)$. A configuration $\gamma' = (q', \alpha')$ *immediately follows* a configuration $\gamma = (q, \alpha)$ by reading $a \in \Sigma$, denoted $\gamma \rightarrow_a \gamma'$, if there exists $(q, a, \phi, \pi, q') \in \delta$ with $\alpha \models \phi$ and $\alpha' = \pi(\alpha)$.

For a string $w = a_1 \cdots a_n$ and two configurations γ and γ', we denote by $\gamma \Rightarrow_w \gamma'$ that $\gamma \rightarrow_{a_1} \cdots \rightarrow_{a_n} \gamma'$. A configuration γ is *reachable* if there exists a string w such that $\gamma_0 \Rightarrow_w \gamma$. A string w is *accepted* by A if $\gamma_0 \Rightarrow_w \gamma_f$ where γ_f is a final configuration. We denote by $L(A)$ the set of strings accepted by A.

A CNFA A is *deterministic* (or a CDFA) if, for every reachable configuration $\gamma = (q, \alpha)$ and for every symbol $a \in \Sigma$, there is at most one transition $(q, a, \phi, \pi, q') \in \delta$ such that $\alpha \models \phi$.

The *size* of a transition θ or acceptance condition $F(q)$ is the number of symbols which occur in it plus the size of the binary representation of each integer occcurring in it. By the same token, the size of A, denoted by $|A|$, is $|Q| + \sum_{q \in Q} \log \tau(q) + |F(q)| + \sum_{\theta \in \delta} |\theta|$.

Theorem 8. *1. Given CNFAs A_1 and A_2, a CNFA A accepting the union or intersection of A_1 and A_2 can be constructed in polynomial time. Moreover, when A_1 and A_2 are deterministic, then so is A.*
2. Given a CDFA A, a CDFA which accepts the complement of A can be constructed in polynomial time.
3. MEMBERSHIP *for word w and CDFA A is in time $\mathcal{O}(|w||A|)$.*
4. MEMBERSHIP *for non-deterministic CNFA is* NP*-complete.*
5. EMPTINESS *for CDFAs and CNFAs is* PSPACE*-complete.*
6. Deciding whether a CNFA A is deterministic is PSPACE*-complete.*

6 From RE(Σ,#) to CNFA

In this section, we show how an RE(Σ,#) expression r can be translated in polynomial time into an equivalent CNFA G_r by applying a natural extension of the well-known Glushkov construction. We emphasize at this point that such an extended Glushkov construction has already been given by Sperberg-McQueen [20]. Therefore, the contribution of this section lies mostly in the characterization given below: G_r is deterministic if and only if r is strongly deterministic. Moreover, as seen in the previous section, CDFAs have desirable properties which by this translation also apply to strongly deterministic RE(Σ,#) expressions. We refer to G_r as the *Glushkov counting automaton of* r.

6.1 Notation and terminology

We first provide some notation and terminology needed in the construction below. For an RE(Σ,#) expression r, the set first(r) (respectively, last(r)) consists of all symbols which are the first (respectively, last) symbols in some word defined by r. These sets are inductively defined as follows:

- $\text{first}(\varepsilon) = \text{last}(\varepsilon) = \emptyset$ and $\forall a \in \text{Char}(r), \text{first}(a) = \text{last}(a) = \{a\}$;

- $\text{first}(r_1 + r_2) = \text{first}(r_1) \cup \text{first}(r_2)$ and $\text{last}(r_1 + r_2) = \text{last}(r_1) \cup \text{last}(r_2)$;
- If $\varepsilon \in L(r_1)$, $\text{first}(r_1 r_2) = \text{first}(r_1) \cup \text{first}(r_2)$, else $\text{first}(r_1 r_2) = \text{first}(r_1)$;
- If $\varepsilon \in L(r_2)$, $\text{last}(r_1 r_2) = \text{last}(r_1) \cup \text{last}(r_2)$, else $\text{last}(r_1 r_2) = \text{last}(r_2)$;
- $\text{first}(r^{k,\ell}) = \text{first}(r_1)$ and $\text{last}(r^{k,\ell}) = \text{last}(r_1)$.

For a regular expression r, we say that a subexpression of r of the form $s^{k,\ell}$ is an *iterator* or *iterated subexpression of* r. Let $\text{lower}(s^{k,\ell}) := k$, and $\text{upper}(s^{k,\ell}) := \ell$. We say that $s^{k,\ell}$ is *bounded* when $\ell \in \mathbb{N}$, otherwise it is *unbounded*. For instance, an iterator of the form $s^{0,\infty}$ is a nullable, unbounded iterator.

For a marked symbol $\overline{x}$ and an iterator c we denote by $\text{iterators}(\overline{x}, c)$ the list of all iterated subexpressions of c which contain $\overline{x}$, except c itself. For marked symbols $\overline{x}, \overline{y}$, we denote by $\text{iterators}(\overline{x}, \overline{y})$ all iterated subexpressions which contain $\overline{x}$ but not $\overline{y}$. Finally, let $\text{iterators}(\overline{x})$ be the list of all iterated subexpressions which contain $\overline{x}$. Note that all such lists $[c_1, \dots, c_n]$ contain a sequence of nested subexpressions. Therefore, we will always assume that they are ordered such that $c_1 \prec c_2 \prec \cdots \prec c_n$. Here $c \prec c'$ denotes that c is a subexpression of c'. For example, if $\overline{r} = ((a_1^{1,2} b_1)^{3,4})^{5,6}$, then $\text{iterators}(a_1, \overline{r}) = [a_1^{1,2}, (a_1^{1,2} b_1)^{3,4}]$, $\text{iterators}(a_1, b_1) = [a_1^{1,2}]$, and $\text{iterators}[a_1] = [a_1^{1,2}, (a_1^{1,2} b_1)^{3,4}, ((a_1^{1,2} b_1)^{3,4})^{5,6}]$.

6.2 Construction

We now define the set $\text{follow}(\overline{r})$ for a marked regular expression $\overline{r}$. As in the standard Glushkov construction, this set lies at the basis of the transition relation of G_r. The set $\text{follow}(\overline{r})$ contains triples $(\overline{x}, \overline{y}, c)$, where $\overline{x}$ and $\overline{y}$ are marked symbols and c is either an iterator or null. Intuitively, the states of G_r will be a designated start state plus a state for each symbol in $\text{Char}(\overline{r})$. A triple $(\overline{x}, \overline{y}, c)$ then contains the information we need for G_r to make a transition from state $\overline{x}$ to $\overline{y}$. If $c \neq \text{null}$, this transition iterates over c and all iterators in $\text{iterators}(\overline{x}, c)$ are reset by going to $\overline{y}$. Otherwise, if c equals null, the iterators in $\text{iterators}(\overline{x}, \overline{y})$ are reset. Formally, the set $\text{follow}(\overline{r})$ contains for each subexpression $\overline{s}$ of $\overline{r}$,

- all tuples $(\overline{x}, \overline{y}, \text{null})$ for $\overline{x}$ in $\text{last}(\overline{s_1})$, $\overline{y}$ in $\text{first}(\overline{s_2})$, and $\overline{s} = \overline{s_1}\,\overline{s_2}$; and
- all tuples $(\overline{x}, \overline{y}, \overline{s})$ for $\overline{x}$ in $\text{last}(\overline{s_1})$, $\overline{y}$ in $\text{first}(\overline{s_1})$, and $\overline{s} = \overline{s_1}^{k,\ell}$.

We introduce a counter variable $\text{cv}(c)$ for every iterator c in $\overline{r}$ whose value will always denote which iteration of c we are doing in the current run on the string. We define a number of tests and update commands on these counter variables:

- $\text{value-test}([c_1, \dots, c_n]) := \bigwedge_{c_i} (\text{lower}(c_i) \leq \text{cv}(c_i)) \wedge (\text{cv}(c_i) \leq \text{upper}(c_i))$. When we leave the iterators $c_1, \dots, c_n$ we have to check that we have done an admissible number of iterations for each iterator.
- $\text{upperbound-test}(c) := \text{cv}(c) < \text{upper}(c)$ when c is a bounded iterator and $\text{upperbound-test}(c) := \text{true}$ otherwise. When iterating over a bounded iterator, we have to check that we can still do an extra iteration.
- $\text{reset}(c_1, \dots, c_n) := \{\text{reset}(\text{cv}(c_1)), \dots, \text{reset}(\text{cv}(c_n))\}$. When leaving some iterators, their values must be reset. The counter variable is reset to 1, because at the time we reenter this iterator, its first iteration is started.

– update(c) := $\{\text{cv}(c){+}{+}\}$. When iterating over an iterator, we start a new iteration and increment its number of transitions.

We now define the Glushkov counting automaton $G_r = (Q, q_0, C, \delta, F, \tau)$. The set of states Q is the set of symbols in $\overline{r}$ plus an initial state, i.e., $Q := \{q_0\} \uplus \bigcup_{\overline{x} \in \text{Char}(\overline{r})} q_{\overline{x}}$. Let C be the set of iterators occurring in $\overline{r}$. We next define the transition function. For all $\overline{y} \in \text{first}(\overline{r})$, $(q_0, \text{dm}(\overline{y}), \mathit{true}, \emptyset, q_{\overline{y}}) \in \delta$.[5] For every element $(\overline{x}, \overline{y}, c) \in \text{follow}(\overline{r})$, we define a transition $(q_{\overline{x}}, \text{dm}(\overline{y}), \phi, \pi, q_{\overline{y}}) \in \delta$. If $c = \text{null}$, then $\phi := \text{value-test}(\text{iterators}(\overline{x}, \overline{y}))$ and $\pi := \text{reset}(\text{iterators}(\overline{x}, \overline{y}))$. If $c \neq \text{null}$, then $\phi := \text{value-test}(\text{iterators}(\overline{x}, c)) \wedge \text{upperbound-test}(c)$ and $\pi := \text{reset}(\text{iterators}(\overline{x}, c)) \cup \text{update}(c)$. The acceptance criteria of G_r depend on the set last($\overline{r}$). For any symbol $\overline{x} \notin \text{last}(\overline{r})$, $F(q_{\overline{x}}) := \text{false}$. For every element $\overline{x} \in \text{last}(\overline{r})$, $F(q_{\overline{x}}) := \text{value-test}(\text{iterators}(\overline{x}))$. Here, we test whether we have done an admissible number of iterations of all iterators in which $\overline{x}$ is located. Finally, $F(q_0) := \text{true}$ if $\varepsilon \in L(r)$. Lastly, for all bounded iterators c, $\tau(\text{cv}(c)) = \text{upper}(c)$ since c never becomes larger than upper(c), and for all unbounded iterators c, $\tau(\text{cv}(c)) = \text{lower}(c)$ as there are no upper bound tests for cv(c).

Theorem 9. *For every RE(Σ,#) expression r, $L(G_r) = L(r)$. Moreover, G_r is deterministic iff r is strongly deterministic.*

7 Decidability and Complexity Results

Definition 3, defining strong determinism, is of a semantical nature. Therefore, we provide Algorithm 1 for testing whether a given expression is strongly deterministic, which runs in cubic time. To decide weak determinism, Kilpeläinen and Tuhkanen [13] give a cubic algorithm for RE(Σ,#), while Brüggemann-Klein [1] gives a quadratic algorithm for RE(Σ) by computing its Glushkov automaton and testing whether it is deterministic[6].

Theorem 10. *For any $r \in$ RE(Σ,#), isStrongDeterministic(r) returns true if and only if r is strong deterministic. Moreover, it runs in time $\mathcal{O}(|r|^3)$.*

We next consider the following decision problems, for expressions of class $\mathcal{R}$:
INCLUSION: Given two expressions $r, r' \in \mathcal{R}$, is $L(r) \subseteq L(r')$?
EQUIVALENCE: Given two expressions $r, r' \in \mathcal{R}$, is $L(r) = L(r')$?
INTERSECTION: Given a number of expressions $r_1, \ldots, r_n \in \mathcal{R}$, is $\bigcap_{i=1}^{n} L(r_i) \neq \emptyset$?

Theorem 11. *(1)* INCLUSION *and* EQUIVALENCE *for RE(Σ,#) are* EXPSPACE-*complete [16],* INTERSECTION *for RE(Σ,#) is* PSPACE-*complete [7]. (2)* INCLUSION *and* EQUIVALENCE *for DET$_W$(Σ) are in* PTIME, INTERSECTION *for DET$_W$(Σ) is* PSPACE-*complete [15]. (3)* INCLUSION *for DET$^{\#}_W$(Σ) is co*NP-*hard [11].*

[5] Recall that dm($\overline{y}$) denotes the demarking of $\overline{y}$.

[6] There sometimes is some confusion about this result: Computing the Glushkov automaton is quadratic in the expression, while linear in the output automaton (consider, e.g., $(a_1 + \cdots + a_n)(a_1 + \cdots + a_n)$). Only when the alphabet is fixed is the Glushkov automaton of a deterministic expression of size linear in the expression.

Algorithm 1 isStrongDeterministic. Returns true if r is strong deterministic, false otherwise.

```
    $\overline{r} \leftarrow$ marked version of $r$
 2: Initialize Follow $\leftarrow \emptyset$
    Compute first($\overline{s}$), last($\overline{s}$), for all subexpressions $\overline{s}$ of $\overline{r}$
 4: if $\exists \overline{x}, \overline{y} \in$ first($\overline{r}$) with $\overline{x} \neq \overline{y}$ and dm($\overline{x}$) = dm($\overline{y}$) then return false
    for each subexpression $\overline{s}$ of $\overline{r}$, in bottom-up fashion do
 6:     if $\overline{s} = \overline{s_1}\,\overline{s_2}$ then
            if last($\overline{s_1}$) $\neq \emptyset$ and $\exists \overline{x}, \overline{y} \in$ first($\overline{s_1}$) with $\overline{x} \neq \overline{y}$ and dm($\overline{x}$) = dm($\overline{y}$) then return false
 8:         $F \leftarrow \{(\overline{x}, \mathrm{dm}(\overline{y})) \mid \overline{x} \in \mathrm{last}(\overline{s_1}), \overline{y} \in \mathrm{first}(\overline{s_2})\}$
        else if $\overline{s} = \overline{s_1}^{[k,\ell]}$, with $\ell \geq 2$ then
10:         if $\exists \overline{x}, \overline{y} \in$ first($\overline{s_1}$) with $\overline{x} \neq \overline{y}$ and dm($\overline{x}$) = dm($\overline{y}$) then return false
            $F \leftarrow \{(\overline{x}, \mathrm{dm}(\overline{y})) \mid \overline{x} \in \mathrm{last}(\overline{s_1}), \overline{y} \in \mathrm{first}(\overline{s_1})\}$
12:     if $F \cap$ Follow $\neq \emptyset$ then return false
        if $\overline{s} = \overline{s_1}\,\overline{s_2}$ or $\overline{s} = \overline{s_1}^{k,\ell}$, with $\ell \geq 2$ and $k < \ell$ then
14:         Follow $\leftarrow$ Follow $\uplus F$
    return true
```

By combining (1) and (2) of Theorem 11 we get the complexity of INTERSECTION for $\mathrm{DET}_W^{\#}(\Sigma)$ and $\mathrm{DET}_S^{\#}(\Sigma)$. This is not the case for the INCLUSION and EQUIVALENCE problem, unfortunately. By using the results of the previous sections we can, for $\mathrm{DET}_S^{\#}(\Sigma)$, give a PSPACE upperbound for both problems, however.

Theorem 12. *(1)* EQUIVALENCE *and* INCLUSION *for $DET_S^{\#}(\Sigma)$ are in* PSPACE. *(2)* INTERSECTION *for $DET_W^{\#}(\Sigma)$ and $DET_S^{\#}(\Sigma)$ is* PSPACE*-complete.*

8 Conclusion

We investigated and compared the notions of strong and weak determinism in the presence of counting. Weakly deterministic expressions have the advantage of being more expressive and more succinct than strongly deterministic ones. However, strongly deterministic expressions are expressivily equivalent to standard deterministic expressions, a class of languages much better understood than the weakly deterministic languages with counting. Moreover, strongly deterministic expressions are conceptually simpler (as strong determinism does not depend on intricate interplays of the counter values) and correspond naturally to deterministic Glushkov automata. The latter also makes strongly deterministic expressions easier to handle as witnessed by the PSPACE upperbound for inclusion and equivalence, whereas for weakly deterministic expressions only a trivial EXPSPACE upperbound is known. For these reasons, one might wonder if the weak determinism demanded in the current standards for XML Schema should not be replaced by strong determinism. The answer to some of the following open questions can shed more light on this issue: (1) Is it decidable if a language is definable by a weakly deterministic expression with counting? (2) Can the Glushkov construction given in Section 6 be extended such that it translates any weakly deterministic expression with counting into a CDFA? (3) What are the exact complexity bounds for inclusion and equivalence of strongly and weakly deterministic expression with counting?

References

1. A. Brüggemann-Klein. Regular expressions into finite automata. *Theor. Comput. Sci.*, 120(2):197–213, 1993.
2. A. Brüggemann-Klein and D. Wood. One-unambiguous regular languages. *Information and Computation*, 142(2):182–206, 1998.
3. D. Colazzo, G. Ghelli, and C. Sartiani. Efficient asymmetric inclusion between regular expression types. In *ICDT*, pages 174–182, 2009.
4. S. Dal-Zilio and D. Lugiez. XML schema, tree logic and sheaves automata. In *RTA*, pages 246–263, 2003.
5. J. Esparza. Decidability and complexity of Petri net problems – an introduction. In *Petri Nets*, pages 374–428, 1996.
6. M.R. Garey and D.S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* Freeman, 1979.
7. W. Gelade, W. Martens, and F. Neven. Optimizing schema languages for XML: Numerical constraints and interleaving. In *ICDT*, pages 269–283, 2007.
8. W. Gelade. Succinctness of regular expressions with interleaving, intersection and counting. In *MFCS*, pages 363–374, 2008.
9. A. Hume. A tale of two greps. *Softw., Pract. and Exp.*, 18(11):1063–1072, 1988.
10. P. Kilpeläinen. Inclusion of unambiguous #res is NP-hard, May 2004. Unpublished.
11. P. Kilpeläinen and R. Tuhkanen. Regular expressions with numerical occurrence indicators — preliminary results. In *SPLST 2003*, pages 163–173, 2003.
12. P. Kilpeläinen and R. Tuhkanen. Towards efficient implementation of XML schema content models. In *DOCENG 2004*, pages 239–241. ACM, 2004.
13. P. Kilpeläinen and R. Tuhkanen. One-unambiguity of regular expressions with numeric occurrence indicators. *Inform. Comput.*, 205(6):890–916, 2007.
14. C. Koch and S. Scherzinger. Attribute grammars for scalable query processing on XML streams. *VLDB Journal*, 16(3):317–342, 2007.
15. W. Martens, F. Neven, and T. Schwentick. Complexity of decision problems for simple regular expressions. In *MFCS*, pages 889–900, 2004.
16. A.R. Meyer and L.J. Stockmeyer. The equivalence problem for regular expressions with squaring requires exponential space. In *FOCS*, p. 125–129, 1972.
17. D.W. Mount. *Bioinformatics: Sequence and Genome Analysis.* Cold Spring Harbor Laboratory Press, September 2004.
18. H. Seidl, T. Schwentick, A. Muscholl, and P. Habermehl. Counting in Trees for Free. In *ICALP*, pages 1136–1149, 2004.
19. G. Pighizzini and J. Shallit. Unary language operations, state complexity and Jacobsthal's function. *Int. J. Found. Comp. Sc.*, 13(1):145–159, 2002.
20. C.M. Sperberg-McQueen. Notes on finite state automata with counters. http://www.w3.org/XML/2004/05/msm-cfa.html, 2004.
21. C.M. Sperberg-McQueen and H. Thompson. XML Schema. http://www.w3.org/XML/Schema, 2005.
22. M.Y. Vardi. From monadic logic to PSL. In *Pillars of Computer Science*, pages 656–681, 2008.
23. L. Wall, T. Christiansen, and J. Orwant. *Programming Perl.* O'Reilly, 2000.