DOCTORAATSPROEFSCHRIFT

2009 | School voor Informatietechnologie
Kennistechnologie, Informatica, Wiskunde, ICT

Foundations of XML: Regular Expressions Revisited

Proefschrift voorgelegd tot het behalen van de graad van
Doctor in de Wetenschappen: richting informatica, te verdedigen door:

Wouter GELADE

Promotor: prof. dr. Frank Neven

Universiteit Maastricht

universiteit hasselt

D/2009/2451/43

Preface

This is the ideal opportunity to thank the people who have contributed either directly or indirectly to this thesis.

First and foremost, I would like to thank my advisor, Frank Neven, for his continuing guidance and support. Next to his excellent vision on scientific work, he also is a great person whose students always come first. Needless to say, this dissertation would not have been possible without him.

During the last years I met many very interesting persons, too many to thank all of them personally. But thank you Thomas, Bart, Wim, Henrik, Walied, Geert Jan, Tim, Marcel, Kurt, Jan, Volker, Natalia, Goele, Stijn, Marc, Wenfei, Floris, ... for pleasant collaborations (scientific as well as teaching), interesting conversations, and good company. Among them, special thanks go out to Wim for many fun collaborations, and Thomas for inviting me to his group in Dortmund, which really felt like a second research group to me.

On a more personal note, I would also like to thank my friends for their continuing support. Last, but definitely not least, I would like to thank my parents, brothers, and sister, for being, each in their own way, simply the best people I know. I can not imagine growing up in a better family as I did.

Thank you all!

Diepenbeek, September 2009

Contents

Preface **i**

1 Introduction **1**

2 Definitions and preliminaries **7**
2.1 Regular Expressions . . . 7
2.2 Deterministic Regular Expressions . . . 9
2.3 Finite Automata . . . 10
2.4 Schema Languages for XML . . . 11
2.5 Decision Problems . . . 12

I Foundations of Regular Expressions **15**

3 Succinctness of Regular Expressions **17**
3.1 A Generalization of a Theorem by Ehrenfeucht and Zeiger to a Fixed Alphabet . . . 19
3.2 Complementing Regular Expressions . . . 27
3.3 Intersecting Regular Expressions . . . 32

4 Succinctness of Extended Regular Expressions **37**
4.1 Definitions and Basic Results . . . 40
4.2 Succinctness w.r.t. NFAs . . . 43
4.3 Succinctness w.r.t. DFAs . . . 44
4.4 Succinctness w.r.t. Regular Expressions . . . 47
4.4.1 $\mathcal{K}_n$: The Basic Language . . . 48
4.4.2 $\mathcal{L}_n$: Binary Encoding of $\mathcal{K}_n$ . . . 49
4.4.3 $\mathcal{M}_n$: Succinctness of RE(&) . . . 52

5 Complexity of Extended Regular Expressions **57**
5.1 Automata for RE(#, &) . . . 57

5.2 Complexity of Regular Expressions 66

6 Deterministic Regular Expressions with Counting 75
6.1 Preliminaries 78
6.2 Expressive Power 80
6.3 Succinctness 89
6.4 Counter Automata 92
6.5 From RE(#) to CNFA 97
6.6 Testing Strong Determinism 104
6.7 Decision Problems for RE(#) 109
6.8 Discussion 110

II Applications to XML Schema Languages 111

7 Succinctness of Pattern-Based Schema Languages 113
7.1 Preliminaries and Basic Results 115
7.1.1 Pattern-Based Schema Languages 116
7.1.2 Characterizations of (Extended) DTDs 117
7.1.3 Theorems 119
7.1.4 Succinctness 120
7.2 Regular Pattern-Based Schemas 120
7.3 Linear Pattern-Based Schemas 126
7.4 Strongly Linear Pattern-Based Schemas 133

8 Optimizing XML Schema Languages 141
8.1 Complexity of DTDs and Single-Type EDTDs 143
8.2 Complexity of Extended DTDs 144
8.3 Simplification 147

9 Handling Non-Deterministic Regular Expressions 151
9.1 Deciding Determinism 155
9.2 Constructing Deterministic Expressions 157
9.2.1 Growing automata 158
9.2.2 Enumerating Automata 158
9.2.3 Optimizing the BKW-Algorithm 161
9.3 Approximating Deterministic Regular Expressions 165
9.3.1 Optimal Approximations 165
9.3.2 Quality of the approximation 166
9.3.3 Ahonen's Algorithm 167
9.3.4 Ahonen's Algorithm Followed by Grow 171
9.3.5 Shrink 171

9.4 Experiments 173
9.4.1 Deciding Determinism 173
9.4.2 Constructing Deterministic Regular Expressions 174
9.4.3 Approximating Deterministic Regular Expressions . . . 175
9.5 SUPAC: Supportive UPA Checker 177

10 Conclusion **179**

11 Publications **181**

Bibliography **183**

Samenvatting **193**

1 Introduction

The regular languages constitute a highly robust class of languages. They can be represented by many different formalisms such as finite automata [92], regular expressions [67], finite semigroups [97], monadic second-order logic [18], right-linear grammars [21], quantifier-free first-order updates [39], ... Regular languages are closed under a wide range of operations and, even more importantly, almost any interesting problem concerning regular languages is decidable. All kinds of aspects of the regular languages have been studied over the past 50 years. From a practical perspective, the most widespread way to specify regular languages is by using regular expressions (REs). They are used in applications in many different areas of computer science, including bioinformatics [84], programming languages [108], model checking [105], and XML schema language [100].

XML is the lingua franca and the de facto standard for data exchange on the Web. When two parties exchange data in XML, the XML documents usually adhere to a certain format. Thereto, XML schema languages are used to describe the structure an XML document can have. The most common XML schema languages are DTD, XML Schema [100], both W3C standards, and Relax NG [22]. From a formal language theory point of view each of these is a grammar-based formalism with regular expressions at their right-hand sides. These expression, however, differ from the standard regular expressions in that they are extended by additional operators but are also restricted by the requirement to be deterministic. Although these requirements are recorded in W3C and ISO standards, it is not clear what their impact on the different schema languages is. The goal of this thesis therefore is a study of these consequences; in particular, we study the complexity of optimization in the presence of additional operators, illustrate the difficulties of migrating from one schema language to another, and study the implications of the determinism constraint (also in the presence of additional operators) and try to make it accessible in practice.

Although the questions we ask are mainly inspired by questions about XML, we believe that the answers are also interesting for the general theoretical computer science community as they answer fundamental questions concerning regular expressions. Therefore, this thesis consists of two parts. In the first part, we study fundamental aspects of regular expression. In the second, we apply the former results to applications concerning XML schema languages. Although most of the work is of a foundational nature, we also developed software to bring these theoretical insights to practice (cf. Chapter 9).

Foundations of Regular Expressions

In **Chapter 3**, we address succinctness of regular expressions. In particular, we consider the following questions. As usual $L(r)$ denotes the language defined by regular expression r. Given regular expressions $r, r_1, \dots, r_k$ over an alphabet Σ,

1. what is the complexity of constructing a regular expression $r_\neg$ defining $\Sigma^* \setminus L(r)$, that is, the complement of r?

2. what is the complexity of constructing a regular expression $r_\cap$ defining $L(r_1) \cap \cdots \cap L(r_k)$?

In both cases, the naive algorithm takes time double exponential in the size of the input. Indeed, for the complement, transform r to an NFA and determinize it (first exponential step), complement it and translate back to a regular expression (second exponential step). For the intersection there is a similar algorithm through a translation to NFAs, taking the crossproduct and a retranslation to a regular expression. Note that both algorithms do not only take double exponential time but also result in a regular expression of double exponential size. We show that these naive constructions can not be substantially improved by exhibiting classes of regular expressions for which this double exponential size increase can not be avoided. The main technical contribution of this chapter is a generalization of a result by Ehrenfeucht and Zeiger [30]. In particular, we construct a family of languages over a two-letter alphabet which can not be defined by small regular expressions. The latter also allows to show that in a translation from automata to regular expression an exponential size-increase can in general not be avoided, even over a binary alphabet.

In addition, we consider the same questions for two strict subclasses: single-occurrence and deterministic regular expressions. A regular expression is single-occurrence when every alphabet symbol occurs at most once. For instance, $(a+b)^*c$ is a single-occurrence regular expression (SORE) while $a^*(a+b)^*$ is not. Despite their apparent simplicity, SOREs nonetheless capture the majority of XML schemas on the Web [9]. Determinism (also called one-unambiguity [17]) intuitively requires that, when matching a string from left to 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 deterministic, but the equivalent expression $b^*a(b^*a)^*$ is. The XML schema languages DTD and XML Schema [100] require expressions to be deterministic. In short, for both classes complementation becomes easier, while intersection remains difficult.

While Chapter 3 investigates the complexity of applying language operations to regular expressions, **Chapter 4** investigates the succinctness of extended regular expressions, i.e. expressions extended with additional operators. In particular, we study the succinctness of regular expressions extended with counting (RE(#)), intersection (RE($\cap$)), and interleaving (RE(&)) operators. The counting operator allows for expressions such as $a^{2,5}$, specifying that there must occur at least two and at most five a's. These RE(#)s are used in egrep [58] and Perl [108] patterns and XML Schema [100]. The class RE($\cap$) is a well studied extension of the regular expressions, and is often referred to as the semi-extended regular expressions. The interleaving operator allows for expressions such as $a \,\&\, b \,\&\, c$, specifying that a, b, and c may occur in any order, and is used in the XML schema language Relax NG [22] and, in a very restricted form, in XML Schema [100]. We give a complete overview of the succinctness of these classes of extended regular expressions with respect to regular expressions, NFAs, and DFAs.

The main reason to study the complexity of a translation from extended regular expressions to standard expressions is to gain more insight in the power of the different operators. However, these results also have important consequences concerning the complexity of translating from one schema language to another. For instance, we show that in a translation from RE(&) to standard RE a double exponential size increase can in general not be avoided. As Relax NG allows interleaving, and XML Schema only allows a very restricted form of interleaving, this implies that also in a translation from Relax NG to XML Schema a double exponential size increase can not be avoided. Nevertheless, as XML Schema is a widespread W3C standard, and Relax NG is a more flexible alternative, such a translation would be more than desirable. Second, we study the succinctness of extended regular expressions to finite automata as, when considering algorithmic problems, the easiest solution is often through a translation to finite automata. However, our negative results indicate that it is often more desirable to develop dedicated algorithms for extended regular expressions which avoid such translations.

In **Chapter 5**, we investigate the complexity of the equivalence, inclusion, and intersection non-emptiness problem for various classes of regular expressions. These classes are general regular expressions extended with counting and interleaving operators, and CHAin Regular Expressions (CHAREs) extended with counting. The latter is another simple subclass of the regular expressions [75]. The main motivation for studying these classes lies in their application to the complexity of the different XML schema languages and, consequently, these results will further be used in Chapter 8, when studying the complexity of the same problems for XML schema languages.

In **Chapter 6**, we study deterministic regular expressions with counting, as these are essentially the regular expressions allowed in XML Schema. The commonly accepted notion of determinism is as given before: an expression is deterministic when matching a string against it, it is always clear against which position in the expression the next symbol must be matched. However, one can also consider a stronger notion of determinism in which it additionally must also always be clear *how* to go from one position to the next. We refer to the former notion as weak and the latter as strong determinism. 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.

For standard regular expressions this distinction is usually not made as the two notions almost coincide for them. That is, a weak deterministic expression can be translated in linear time into an equivalent strong deterministic one [15].[1] This situation changes completely when counting is involved. Firstly, the algorithm for deciding whether an expression is weakly deterministic is non-trivial [66]. For instance, $(a^{2,3} + b)^{2,2}b$ is weakly deterministic, but the very similar $(a^{2,3} + b)^{3,3}b$ is not. So, the amount of non-determinism introduced depends on the concrete values of the counters. Second, as we show, weakly deterministic expressions with counting are strictly more expressive than strongly deterministic ones. Therefore, the aim of this chapter is an in-depth study of the notions of weak and strong determinism in the presence of counting with respect to expressiveness, succinctness, and complexity.

[1] Brüggemann-Klein [15] did not study strong determinism explicitly. However, she does give a procedure to transform expressions into *star normal form* which rewrites weakly determinisistic expressions into equivalent strongly deterministic ones in linear time.

Applications to XML Schema Languages

In **Chapter 7**, we study pattern-based schema languages. This is a schema language introduced by Martens et al. [77] and equivalent in expressive power to single-type EDTDs, the commonly used abstraction of XML Schema. An advantage of this language is that it makes the expressiveness of XML Schema more apparent: the content model of an element can only depend on regular string properties of the string formed by the ancestors of that element. Pattern-based schemas can therefore be used as a type-free front-end for XML Schema. As they can be interpreted both in an existential and universal way, we study in this chapter the complexity of translating between the two semantics and into the formalisms of DTDs, EDTDs, and single-type EDTDs, the common abstractions of DTD, Relax NG, and XML Schema, respectively

Here, we make extensive use of the results in Chapter 3 and show that in general translations from pattern-based schemas to the other schema formalisms requires exponential or even double exponential time, thereby reducing much of the hope of using pattern-based schemas in its most general form as a useful front-end for XML Schema. Therefore, we also study more restricted classes of schemas: linear and strongly linear pattern-based schemas. Interestingly, strongly linear schemas, the most restricted class, allow for efficient translations to other languages, efficient algorithms for basic decision problems, and yet are expressive enough to capture the far majority of real-world XML Schema Definitions (XSDs). From a practical point of view, this is hence a very useful class of schemas.

In **Chapter 8**, we study the impact of adding counting and interleaving to regular expressions for the different schema languages. We consider the equivalence, inclusion, and intersection non-emptiness problem as these constitute the basic building blocks in algorithms for optimizing XML schemas. We also consider the simplification problem: Given an EDTD, is it equivalent to a single-type EDTD or a DTD?

Finally, in **Chapter 9**, we provide algorithms for dealing with the requirement in DTD and XML Schema that all regular expressions are deterministic, i.e. the Unique Particle Attribution constraint in XML Schema. In most books (c.f. [103]), the UPA constraint is usually explained in terms of a simple example rather than by means of a clear syntactical definition. So, when after the schema design process, one or several regular expressions are rejected by the schema checker on account of being non-deterministic, it is very difficult for non-expert[2] users to grasp the source of the error and almost impossible to rewrite the expression into an admissible one. The purpose of the present chapter is to investigate methods for transforming nondeterministic expres-

[2]In formal language theory.

sions into concise and readable deterministic ones defining either the same language or constituting good approximations. We propose the algorithm SUPAC (Supportive UPA Checker) which can be incorporated in a responsive XSD tester which in addition to rejecting XSDs violating UPA also suggests plausible alternatives. Consequently, the task of designing an XSD is relieved from the burden of the UPA restriction and the user can focus on designing an accurate schema.

2 Definitions and preliminaries

In this section, we define the necessary definitions and notions concerning regular expressions, finite automata and XML schema languages.

2.1 Regular Expressions

By $\mathbb{N}$ we denote the natural numbers without zero. For the rest of this thesis, Σ always denotes a finite alphabet. A Σ-*string* (or simply string) is a finite sequence $w = a_1 \cdots a_n$ of Σ-symbols. We define the length of w, denoted by $|w|$, to be n. We denote the empty string by ε. The set of *positions of* w is $\{1, \dots, n\}$ and the *symbol of* w *at position* i is a_i. By $w_1 \cdot w_2$ we denote the *concatenation* of two strings w_1 and w_2. As usual, for readability, we denote the concatenation of w_1 and w_2 by w_1w_2. The set of all strings is denoted by Σ^*. A *string language* is a subset of Σ^*. For two string languages $L, L' \subseteq \Sigma^*$, we define their concatenation $L \cdot L'$ to be the set $\{ww' \mid w \in L, w' \in L'\}$. We abbreviate $L \cdot L \cdots L$ (i times) by L^i. By $w_1 \,\&\, w_2$ we denote the set of strings that is obtained by *interleaving* w_1 and w_2 in every possible way. That is, for $w \in \Sigma^*$, $w \,\&\, \varepsilon = \varepsilon \,\&\, w = \{w\}$, and $aw_1 \,\&\, bw_2 = (\{a\}(w_1 \,\&\, bw_2)) \cup (\{b\}(aw_1 \,\&\, w_2))$. The operator & is then extended to languages in the canonical way.

The set of *regular expressions* over Σ, denoted by RE, is defined in the usual way: $\emptyset$, ε, and every Σ-symbol is a regular expression; and when r_1 and r_2 are regular expressions, then so are $r_1 \cdot r_2$, $r_1 + r_2$, and r_1^*. By RE$(\&, \cap, \neg, \#)$ we denote the class of *extended regular expressions*, that is, REs extended with interleaving, intersection, complementation, and counting operators. So, when

r_1 and r_2 are RE($\&, \cap, \neg, \#$) expressions then so are $r_1 \& r_2$, $r_1 \cap r_2$, $\neg r_1$ and $r_1^{k,\ell}$ for $k \in \mathbb{N} \cup \{0\}$, $\ell \in \mathbb{N} \cup \{\infty\}$ and $k \leq \ell$. Here, $k < \infty$ for any $k \in \mathbb{N} \cup \{0\}$. For any subset S of $\{\&, \cap, \neg, \#\}$, we denote by RE(S) the class of extended regular expressions using only operators in S. For instance, RE($\cap, \neg$) denotes the set of all expressions using, next to the standard operators, only intersection and complementation. To distinguish from extended regular expressions, we often refer to the normal regular expressions as *standard* regular expressions. All notions defined for extended regular expressions in the remainder of this section also apply to standard regular expressions, in the canonical manner.

The language defined by an extended regular expression r, denoted by $L(r)$, is inductively defined as follows:

- $L(\emptyset) = \emptyset$;
- $L(\varepsilon) = \{\varepsilon\}$;
- $L(a) = \{a\}$;
- $L(r_1 r_2) = L(r_1) \cdot L(r_2)$;
- $L(r_1 + r_2) = L(r_1) \cup L(r_2)$;
- $L(r^*) = \{\varepsilon\} \cup \bigcup_{i=1}^{\infty} L(r)^i$;
- $L(r_1 \,\&\, r_2) = L(r_1) \,\&\, L(r_2)$
- $L(r_1 \cap r_2) = L(r_1) \cap L(r_2)$;
- $L(\neg r_1) = \Sigma^* \setminus L(r_1)$; and
- $L(r^{k,\ell}) = \bigcup_{i=k}^{\ell} L(r)^i$.

By r^+, $\bigcup_{i=1}^{k} r_i$, $r?$, and r^k, with $k \in \mathbb{N}$, we abbreviate the expressions rr^*, $r_1 + \cdots + r_k$, $r + \varepsilon$, and $rr \cdots r$ (k-times), respectively. For a set $S = \{a_1, \ldots, a_n\} \subseteq \Sigma$, we abbreviate by S the regular expression $a_1 + \cdots + a_n$. When $r^{k,l}$ is used in a standard regular expression, this is an abbreviation for $r^k(r + \varepsilon)^{l-k}$, unless mentioned otherwise. Note also that in the context of RE($\#$), r^* is an abbreviation of $r^{0,\infty}$. Therefore, we sometimes omit $*$ as a basic operator when counting is allowed. Note also that, in the absence of complementation, the operator $\emptyset$ is only necessary to define the language $\emptyset$ and can be removed in linear time from any extended regular expression defining a language different from $\emptyset$. Therefore, we assume in the remainder of this thesis that the operator $\emptyset$ is only used in the expression $r = \emptyset$, and not in any other expression not containing complementation.

For an extended regular expression r, we denote by $\text{Char}(r)$ the set of Σ-symbols occurring in r. We define the *size* of an extended regular expression r over Σ, denoted by $|r|$, as the number of Σ-symbols and operators occurring in r plus the sizes of the binary representations of the integers. This is equivalent to the length of its (parenthesis-free) reverse Polish form [112]. Formally, $|\emptyset| = |\varepsilon| = |a| = 1$, for $a \in \Sigma$, $|r_1 r_2| = |r_1 \cap r_2| = |r_1 + r_2| = |r_1 \,\&\, r_2| = |r_1 \cap r_2| = |r_1| + |r_2| + 1$, $|r^*| = |\neg r| = |r| + 1$, and $|r^{k,\ell}| = |r| + \lceil \log k \rceil + \lceil \log \ell \rceil$.

Other possibilities considered in the literature for defining the size of a regular expression are: (1) counting all symbols, operators, and parentheses [2, 59]; or, (2) counting only the Σ-symbols. However, it is known (see, for instance [31]) that for standard regular expressions, provided they are preprocessed by syntactically eliminating superfluous $\emptyset$- and ε-symbols, and nested stars, the three length measures are identical up to a constant multiplicative factor. For extended regular expressions, counting only the Σ-symbols is not sufficient, since for instance the expression $(\neg\varepsilon)(\neg\varepsilon)(\neg\varepsilon)$ does not contain any Σ-symbols. Therefore, we define the size of an expression as the length of its reverse Polish form.

For an RE(#) expression r, the set $\text{first}(r)$ (respectively, $\text{last}(r)$) consists of all symbols which are the first (respectively, last) symbols in some string defined by r. These sets are inductively defined as follows:

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

2.2 Deterministic Regular Expressions

As mentioned in the introduction, several XML schema languages restrict regular expressions occurring in rules to be *deterministic* (also sometimes called one-unambiguous [17]). We introduce this notion in this section.

A *marked regular expression* 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. Formally, $\overline{r} \in \text{MRE}$ if $\text{Char}(\overline{r}) \subset \Sigma \times \mathbb{N}$ and, for every subexpression $\overline{s}\overline{s}'$ or $\overline{s} + \overline{s}'$ of $\overline{r}$, $\text{Char}(\overline{s}) \cap \text{Char}(\overline{s}') = \emptyset$. 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 $\mathrm{dm}(\overline{r})$, where $\mathrm{dm} : \mathrm{MRE} \to \mathrm{RE}$ is defined as $\mathrm{dm}(\varepsilon) := \varepsilon$, $\mathrm{dm}((a, i)) := a$, $\mathrm{dm}(\overline{rs}) := \mathrm{dm}(\overline{r})\mathrm{dm}(\overline{s})$, $\mathrm{dm}(\overline{r+s}) := \mathrm{dm}(\overline{r}) + \mathrm{dm}(\overline{s})$, and $\mathrm{dm}(\overline{r^{k,\ell}}) := \mathrm{dm}(\overline{r})^{k,\ell}$. Any function $\mathrm{m} : \mathrm{RE} \to \mathrm{MRE}$ such that for every $r \in \mathrm{RE}$ it holds that $\mathrm{dm}(\mathrm{m}(r)) = r$ 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 $(a + b)a + bc$ is $(a_1 + b_1)a_2 + b_2c_1$. The markings and demarkings of strings are defined analogously. For the rest of the thesis, we usually leave the actual marking and demarking functions m and dm implicit and denote by $\overline{r}$ a marking of the expression r and, conversely, by r the demarking of $\overline{r}$. Likewise $\overline{w}$ will denote a marking of a string w. We always use overlined letters to denote marked expressions, symbols, and strings.

Definition 1. An expression $r \in \mathrm{RE}$ is *deterministic* (also called *one-unambiguous*) if, for all strings $\overline{u}, \overline{v}, \overline{w} \in \mathrm{Char}(\overline{r})^*$ and all symbols $\overline{a}, \overline{b} \in \mathrm{Char}(\overline{r})$, the conditions $\overline{uav}, \overline{ubw} \in L(\overline{r})$ and $\overline{a} \neq \overline{b}$ imply that $a \neq b$.

Intuitively, an expression is 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 string. For instance, $(a + b)^*a$ is not deterministic, while $(b^*a)(b^*a)^*$ is.

2.3 Finite Automata

A non-deterministic finite automaton (NFA) A is a 4-tuple (Q, q_0, δ, F) where Q is the set of states, q_0 is the initial state, F is the set of final states and $\delta \subseteq Q \times \Sigma \times Q$ is the transition relation. By δ^* we denote the reflexive-transitive closure of δ, i.e. δ^* is the smallest relation such that for all $q \in Q$, $(q, \varepsilon, q) \in \delta^*$, $\delta \subset \delta^*$ and when (q_1, u, q_2) and (q_2, v, q_3) are in δ^* then so is (q_1, uv, q_3). So, w is *accepted* by A if $(q_0, w, q') \in \delta^*$ for some $q' \in F$. We also sometimes write $q \Rightarrow_{A,w} q'$ when $(q, w, q') \in \delta^*$. The set of strings accepted by A is denoted by $L(A)$. The size of an NFA is $|\delta|$. An NFA is *deterministic* (or a DFA) if for all $a \in \Sigma, q \in Q$, it holds that $|\{(q, a, q') \in \delta \mid q' \in Q\}| \leq 1$. When A is a DFA we sometimes write δ and δ^* as a function instead of a relation, i.e. $\delta(q, a) = p$ when $(q, a, p) \in \delta$. For a transition $(q, a, p) \in \delta$, we say that q is its *source state* and p its *target state*.

A state $q \in Q$ is *useful* if there exist strings $w, w' \in \Sigma^*$ such that $(q_0, w, q) \in \delta^*$, and $(q, w', q_f) \in \delta^*$, for some $q_f \in F$. An NFA is *trimmed* if it only contains useful states. For $q \in Q$, let $\text{symbols}(q) = \{a \mid \exists p \in Q, (p, a, q) \in \delta\}$. Then, A is *state-labeled* if for any $q \in Q$, $|\text{symbols}(q)| \leq 1$, i.e., all transitions to a single state are labeled with the same symbol. In this case, we also denote this symbol by $\text{symbol}(q)$. Further, A is *non-returning* if $\text{symbols}(q_0) = \emptyset$, i.e., q_0 has no incoming transitions.

We will make use of the following well-known results concerning finite automata and regular expressions.

Theorem 2. *Let A be an NFA over Σ with m states, and let $|A| = n$ and $|\Sigma| = k$.*

1. *A regular expression r, with $L(r) = L(A)$, of size $\mathcal{O}(mk4^m)$, can be constructed in time $2^{\mathcal{O}(n)}$ [82, 31].*
2. *A DFA B with 2^m states, such that $L(B) = L(A)$, can be constructed in time $\mathcal{O}(2^n)$ [110].*
3. *A DFA B with 2^m states, such that $L(B) = \Sigma^* \backslash L(A)$, can be constructed in time $\mathcal{O}(2^n)$ [110].*
4. *Let $r \in \text{RE}$. An NFA B with $|r| + 1$ states, such that $L(B) = L(r)$, can be constructed in time $\mathcal{O}(|r| \cdot k)$ [15].*

2.4 Schema Languages for XML

We use unranked trees as an abstraction for XML documents. The set of *unranked Σ-trees*, denoted by $\mathcal{T}_\Sigma$, is the smallest set of strings over Σ and the parenthesis symbols "(" and ")" containing ε such that, for $a \in \Sigma$ and $w \in (\mathcal{T}_\Sigma)^*$, $a(w)$ is in $\mathcal{T}_\Sigma$. So, a tree is either ε (empty) or is of the form $a(t_1 \cdots t_n)$ where each t_i is a tree. In the tree $a(t_1 \cdots t_n)$, the subtrees $t_1, \ldots, t_n$ are attached to the root labeled a. We write a rather than $a()$. Notice that there is no a priori bound on the number of children of a node in a Σ-tree; such trees are therefore *unranked.* For every $t \in \mathcal{T}_\Sigma$, the *set of nodes* of t, denoted by $\text{Dom}(t)$, is the set defined as follows: *(i)* if $t = \varepsilon$, then $\text{Dom}(t) = \emptyset$; and *(ii)* if $t = a(t_1 \cdots t_n)$, where each $t_i \in \mathcal{T}_\Sigma$, then $\text{Dom}(t) = \{\varepsilon\} \cup \bigcup_{i=1}^{n} \{iu \mid u \in \text{Dom}(t_i)\}$. For a node $u \in \text{Dom}(t)$, we denote the label of u by $\text{lab}^t(u)$. In the sequel, whenever we say tree, we always mean Σ-tree. We denote by $\text{anc-str}^t(u)$ the sequence of labels on the path from the root to u including both the root and u itself, and $\text{ch-str}^t(u)$ denotes the string formed by the labels of the children of u, i.e., $\text{lab}^t(u1) \cdots \text{lab}^t(un)$. Denote by $t_1[u \leftarrow t_2]$ the tree obtained from a tree t_1 by replacing the subtree rooted at node u of t_1 by t_2. By $\text{subtree}^t(u)$ we denote the subtree of t rooted at u. A *tree language* is a set of trees.

We make use of the following definitions to abstract from the commonly used schema languages [77]:

Definition 3. Let $\mathcal{R}$ be a class of representations of regular string languages over Σ.

1. A *DTD($\mathcal{R}$)* over Σ is a tuple (Σ, d, s_d) where d is a function that maps Σ-symbols to elements of $\mathcal{R}$ and $s_d \in \Sigma$ is the start symbol. For notational convenience, we sometimes denote (Σ, d, s_d) by d and leave the start symbol s_d and alphabet Σ implicit.

   A tree t *satisfies* d if *(i)* $\text{lab}^t(\varepsilon) = s_d$ and, *(ii)* for every $u \in \text{Dom}(t)$ with n children, $\text{lab}^t(u1) \cdots \text{lab}^t(un) \in L(d(\text{lab}^t(u)))$. By $L(d)$ we denote the set of trees satisfying d.

2. An *extended DTD* (EDTD($\mathcal{R}$)) over Σ is a 5-tuple $D = (\Sigma, \Sigma', d, s, \mu)$, where Σ' is an alphabet of *types*, (Σ', d, s) is a DTD($\mathcal{R}$) over Σ', and μ is a mapping from Σ' to Σ.

   A tree t then *satisfies* an extended DTD if $t = \mu(t')$ for some $t' \in L(d)$. Here we abuse notation and let μ also denote its extension to define a homomorphism on trees. Again, we denote by $L(D)$ the set of trees satisfying D. For ease of exposition, we always take $\Sigma' = \{a^i \mid 1 \leq i \leq k_a, a \in \Sigma, i \in \mathbb{N}\}$ for some natural numbers k_a, and we set $\mu(a^i) = a$.

3. A *single-type EDTD* (EDTD$^{\text{st}}$($\mathcal{R}$)) over Σ is an EDTD($\mathcal{R}$) $D = (\Sigma, \Sigma', d, s, \mu)$ with the property that for every $a \in \Sigma'$, in the regular expression $d(a)$ no two types b^i and b^j with $i \neq j$ occur.

We denote by EDTD (respectively, EDTD$^{\text{st}}$) the class EDTD(RE) (respectively, EDTD$^{\text{st}}$(RE)). Similarly, for a subset S of $\{\&, \cap, \neg, \#\}$ we denote by EDTD(S) and EDTD$^{\text{st}}$(S) the classes EDTD(RE(S)) and EDTD$^{\text{st}}$(RE(S)), respectively. As explained in [77, 85], EDTDs and single-type EDTDs correspond to Relax NG and XML Schema, respectively. Furthermore, EDTDs correspond to the unranked regular languages [16], while single-type EDTDs form a strict subset thereof [77].

2.5 Decision Problems

The following decision problems will be studied for various classes of regular expressions, automata, and XML schema languages.

Definition 4. Let $\mathcal{M}$ be a class of representations of regular string or tree languages:

- INCLUSION for $\mathcal{M}$: Given two elements $e, e' \in \mathcal{M}$, is $L(e) \subseteq L(e')$?
- EQUIVALENCE for $\mathcal{M}$: Given two elements $e, e' \in \mathcal{M}$, is $L(e) = L(e')$?
- INTERSECTION for $\mathcal{M}$: Given an arbitrary number of elements $e_1, \ldots, e_n \in \mathcal{M}$, is $\bigcap_{i=1}^{n} L(e_i) \neq \emptyset$?
- MEMBERSHIP for $\mathcal{M}$: Given an element $e \in \mathcal{M}$ and a string or a tree f, is $f \in L(e)$?
- SATISFIABILITY for $\mathcal{M}$: Given an element $e \in \mathcal{M}$, does there exist a non-empty string or tree f such that $f \in L(e)$?
- SIMPLIFICATION for $\mathcal{M}$: Given an element $e \in \mathcal{M}$, does there exist a DTD D such that $L(e) = L(D)$?

Part I

Foundations of Regular Expressions

3

Succinctness of Regular Expressions

The two central questions addressed in this chapter are the following. Given regular expressions (REs) $r, r_1, \ldots, r_k$ over an alphabet Σ,

1. what is the complexity of constructing a regular expression $r_\neg$ defining $\Sigma^* \setminus L(r)$, that is, the complement of r?
2. what is the complexity of constructing a regular expression $r_\cap$ defining $L(r_1) \cap \cdots \cap L(r_k)$?

In both cases, the naive algorithm takes time double exponential in the size of the input. Indeed, for the complement, transform r to an NFA and determinize it (first exponential step), complement it and translate back to a regular expression (second exponential step). For the intersection there is a similar algorithm through a translation to NFAs, taking the crossproduct and a retranslation to a regular expression. Note that both algorithms do not only take double exponential time but also result in a regular expression of double exponential size. In this chapter, we exhibit classes of regular expressions for which this double exponential size increase cannot be avoided. Furthermore, when the number k of regular expressions is fixed, $r_\cap$ can be constructed in exponential time and we prove a matching lower bound for the size increase. In addition, we consider the fragments of deterministic and single-occurrence regular expressions relevant to XML schema languages [9, 11, 43, 77]. Our main results are summarized in Table 3.1.

	complement	intersection (fixed)	intersection (arbitrary)
regular expression	2-exp	exp	2-exp
deterministic	poly	exp	2-exp
single-occurrence	poly	exp	exp

Table 3.1: Overview of the size increase for the various operators and subclasses.

The main technical part of the chapter is centered around the generalization of a result in [30]. They exhibit a class of languages $(\mathcal{K}_n)_{n\in\mathbb{N}}$ each of which can be accepted by a DFA of size $\mathcal{O}(n^2)$ but cannot be defined by a regular expression of size smaller than 2^{n-1}. The most direct way to define $\mathcal{K}_n$ is by the DFA that accepts it: it consists of n states, labeled 0 to $n-1$, where 0 is the initial and $n-1$ the final state, and which is fully connected with the edge between state i and j carrying the label $a_{i,j}$. Note that the alphabet over which $\mathcal{K}_n$ is defined grows quadratically with n. We generalize their result to a fixed alphabet. In particular, we define $\mathcal{H}_n$ as the binary encoding of $\mathcal{K}_n$ using a suitable encoding for $a_{i,j}$ and prove that every regular expression defining $\mathcal{H}_n$ should be at least of size 2^n. As integers are encoded in binary the complement and intersection of regular expressions can now be used to separately encode $\mathcal{H}_{2^n}$ (and slight variations thereof) leading to the desired results.

Although the succinctness of various automata models have been investigated in depth [46] and more recently those of logics over (unary alphabet) strings [47], the succinctness of regular expressions had, up to recently, hardly been addressed. For the complement of a regular expression an exponential lower bound is given in [31]. For the intersection of an arbitrary number of regular expressions Petersen gave an exponential lower bound [90], while [31] mentions a quadratic lower bound for the intersection of two regular expressions. In fact, in [31], it is explicitly asked what the maximum achievable size increase is for the complement of one and the intersection of two regular expressions (Open Problems 4 and 5), and whether an exponential size increase in the translation from DFA to RE is also unavoidable when the alphabet is fixed (Open Problem 3).

More recently, there have been a number of papers concerning succinctness of regular expressions and related matters [53, 48, 49, 50, 51, 52]. Most related is [48], where, independently, a number of problems similar to the problems in this chapter are studied. They also show that in constructing a regular expression for the intersection of two expressions, an exponential size-increase can not be avoided. However, we only give a $2^{\Omega(\sqrt{n})}$ lower bound whereas they obtain $2^{\Omega(n)}$, which in [49] is shown to be almost optimal. For the complementation of an RE we both obtain a double exponential lower bound. Here, however, they obtain $2^{2^{\Omega(\sqrt{n \log n})}}$ whereas we prove a (tight) $2^{2^{\Omega(n)}}$ lower bound. However, in [51] they also prove a $2^{2^{\Omega(n)}}$ lower bound. Finally, as a corollary of our results we obtain that in a translation from a DFA to an RE an exponential size increase can not be avoided, also when the alphabet is fixed. This yields a lower bound of $2^{\Omega(\sqrt{n/\log n})}$ which in [48] is improved to the (tight) bound of $2^{\Omega(n)}$. All results mentioned here involve fixed binary alphabets, and, together, these results settle open problems 3, 4, and 5 of [31].

As already mentioned in the introduction, the main motivation for this research stems from its application in the emerging area of XML-theory [72, 86, 98, 106]. The lower bounds presented here are utilized in Chapter 7 to prove, among other things, lower bounds on the succinctness of existential and universal pattern-based schemas on the one hand, and single-type EDTDs (a formalization of XSDs) and DTDs, on the other hand. As the DTD and XML Schema specification require regular expressions occurring in rules to be deterministic, formalized by Brüggemann-Klein and Wood in terms of one-unambiguous regular expressions [17], we also investigate the complement and intersection of those. In particular, we show that a deterministic regular expressions can be complemented in polynomial time, whereas the lower bounds concerning intersection carry over from unrestricted regular expressions. A study in [9] reveals that most of the deterministic regular expression used in practice take a very simple form: every alphabet symbol occurs at most once. We refer to those as single-occurrence regular expressions (SOREs) and show a tight exponential lower bound for intersection.

Outline. In **Section 3.1** we extend the results of Ehrenfeucht and Zeiger to languages over a fixed alphabet. Then, in **Sections 3.2** and **3.3** we investigate the succinctness of the complement and intersection of regular expressions, respectively.

3.1 A Generalization of a Theorem by Ehrenfeucht and Zeiger to a Fixed Alphabet

We first introduce the family $(\mathcal{K}_n)_{n \in \mathbb{N}}$ of string languages defined by Ehrenfeucht and Zeiger [30] over an alphabet whose size grows quadratically with the parameter n:

Definition 5. Let $n \in \mathbb{N}$ and $\Sigma_n = \{a_{i,j} \mid 0 \leq i, j \leq n-1\}$. Then, $\mathcal{K}_n$ contains exactly all strings of the form $a_{0,i_1} a_{i_1,i_2} \cdots a_{i_{k-1},n-1}$ where $k \in \mathbb{N}$.

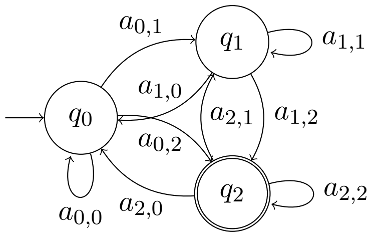


Figure 3.1: The DFA $A_3^{\mathcal{K}}$, accepting $\mathcal{K}_3$.

A way to interpret $\mathcal{K}_n$ is to consider the DFA with states $\{q_0, \dots, q_{n-1}\}$ which is fully connected, whose start state is state q_0 and final state is q_{n-1} and where the edge between state q_i and q_j is labeled with $a_{i,j}$. Formally, let $A_n^{\mathcal{K}} = (Q, q_0, \delta, F)$ be defined as $Q = \{q_0, \dots, q_{n-1}\}$, $F = \{q_{n-1}\}$, and for all $i, j \in [0, n-1], (q_i, a_{i,j}, q_j) \in \delta$. Figure 3.1 shows $A_3^{\mathcal{K}}$.

Ehrenfeucht and Zeiger obtained the succinctness of DFAs with respect to regular expressions through the following theorem:

Theorem 6 ([30])**.** For $n \in \mathbb{N}$, any regular expression defining $\mathcal{K}_n$ must be of size at least 2^{n-1}. Furthermore, there is a DFA of size $\mathcal{O}(n^2)$ accepting $\mathcal{K}_n$.

We will now define the language $\mathcal{H}_n$ as a binary encoding of $\mathcal{K}_n$ over a four-letter alphabet, and generalize Theorem 6 to $\mathcal{H}_n$. Later, we will define $\mathcal{H}_n^2$ as an encoding of $\mathcal{H}_n$ over a binary alphabet.

The language $\mathcal{H}_n$ will be defined as the straightforward binary encoding of $\mathcal{K}_n$ that additionally swaps the pair of indices in every symbol $a_{i,j}$. Thereto, for $a_{i,j} \in \Sigma_n$, define the function ρ_n as

$$\rho_n(a_{i,j}) = \text{enc}(j)\$\text{enc}(i)\#,$$

where enc(i) and enc(j) denote the $\lceil \log(n) \rceil$-bit binary encodings of i and j, respectively. Note that since $i, j < n$, i and j can be encoded using only $\lceil \log(n) \rceil$-bits. We extend the definition of ρ_n to strings in the usual way: $\rho_n(a_{i_0,i_1} \cdots a_{i_{k-1},i_k}) = \rho_n(a_{i_0,i_1}) \cdots \rho_n(a_{i_{k-1},i_k})$.

We are now ready to define $\mathcal{H}_n$.

Definition 7. Let $\Sigma_{\mathcal{H}} = \{0, 1, \$, \#\}$. For $n \in \mathbb{N}$, let $\mathcal{H}_n = \{\rho_n(w) \mid w \in \mathcal{K}_n\}$.

For instance, for $n = 5$, $w = a_{0,2}a_{2,1}a_{1,4}a_{4,4} \in \mathcal{K}_5$ and thus

$$\rho_n(w) = 010\$000\#001\$010\#100\$001\#100\$100\# \in \mathcal{H}_5.$$

Remark 8. Although it might seem a bit artificial to swap the indices in the binary encoding of $\mathcal{K}_n$, it is definitely necessary. Indeed, consider the

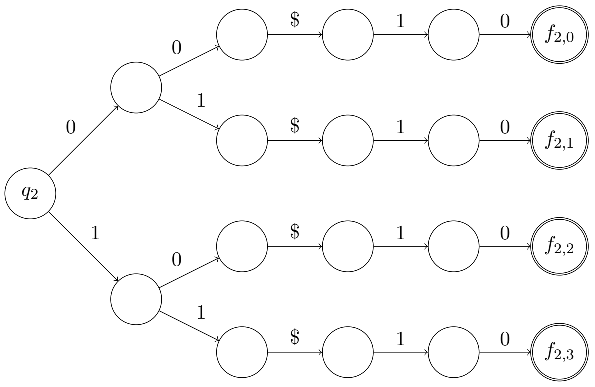


Figure 3.2: The automaton B_2, for $n = 4$.

language $\mathcal{H}'_n$ obtained from $\mathcal{K}_n$ like $\mathcal{H}_n$ but without swapping the indices of the symbols in $\mathcal{K}_n$. This language can be defined by a regular expressions of size $\mathcal{O}(n \log(n))$:

$$\text{enc}(0)\$\big(\bigcup_{i<n} \text{enc}(i)\#\text{enc}(i)\$\big)^{*}\text{enc}(n-1)\#.$$

Therefore, $\mathcal{H}'_n$ is not suited to show exponential or double exponential lower bounds on the size of regular expressions.

We now show that, by using the encoding that swaps the indices, it is possible to generalize Theorem 6 to a four-letter alphabet as follows:

Theorem 9. For any $n \in \mathbb{N}$, with $n \geq 2$,

1. there is a DFA A_n of size $\mathcal{O}(n^2 \log n)$ defining $\mathcal{H}_n$; and,
2. any regular expression defining $\mathcal{H}_n$ is of size at least 2^n.

We first show (1). Let $n \in \mathbb{N}$ and $n \geq 2$. We compose A_n from a number of subautomata which each will be able to read one block of a string. That is, strings defined by the regular expression $(0+1)^{\lceil \log n \rceil}\$(0+1)^{\lceil \log n \rceil}$. For any $i < n$, let $B_i = (Q_i, q_i, \delta_i, \{f_{i,0}, \dots, f_{i,n-1}\})$, with $L(B_i) = \{w\$w' \mid w, w' \in (0+1)^{\lceil \log n \rceil} \wedge \text{enc}(w') = i\}$, such that for any $j < n$, $q \Rightarrow_{B_i, w\$w'} f_{i,j}$ whenever $\text{enc}(w) = j$. That is, B_i reads strings of the form $w\$w'$, checks whether w' encodes i and remembers the value of w in its accepting state. The construction of these automata is straightforward. We illustrate the construction of B_2, for $n = 4$, in Figure 3.2.

Now, let $A_n = (Q, q_0, \delta, q_{n-1})$ where $Q = \bigcup_{i<n} Q_i$ and δ contains $\bigcup_{i<n} \delta_i$ plus for every $i, j < n$, $(f_{i,j}, \#, q_j) \in \delta$. So, A_n works as follows: first A_0

reads the first block of the string, checks whether the second integer is 0 and remembers the first integer of the block, say j. Then it passes control to B_j which reads the next block in which it remembers the first integer, say k, and checks whether the second integer is j. If so, it passes control to B_k, and so on. Whenever A_n reaches the initial state of B_{n-1}, a valid string has been read and A_n can accept.

Finally, for the size of A_n, consider the subautomata B_i as illustrated in Figure 3.2. The first, tree-like, part consists of at most $n + n/2 + n/4 + \cdots + 1$ nodes and transitions, which is bounded by $2n$. Further, the *linear* automata following this part are each of size $\lceil \log n \rceil$ and there are n of these. So, the total size of any B_i is $\mathcal{O}(n \log n)$. Since A_n consists of a linear number of B_i subautomata, A_n is of size $\mathcal{O}(n^2 \log n)$. This concludes the proof of Theorem 9(1).

We now prove Theorem 9(2). It follows the structure of the proof of Ehrenfeucht and Zeiger but is technically more involved as it deals with binary encodings of integers.

We start by introducing some terminology. We say that a language L *covers* a string w if there exist strings $u, u' \in \Sigma^*$ such that $uwu' \in L$. A regular expression r covers w when $L(r)$ covers w. Let $w = a_{i_0,i_1} a_{i_1,i_2} \cdots a_{i_{k-1},i_k}$ be a string covered by $\mathcal{K}_n$. We say that i_0 is the *start-point* of w and i_k is its *end-point*. Furthermore, we say that w *contains* i or i *occurs in* w if i occurs as an index of some symbol in w. That is, $a_{i,j}$ or $a_{j,i}$ occurs in w for some j. For instance, $a_{0,2}a_{2,2}a_{2,1}$ has start-point 0, end-point 1, and contains 0, 1 and 2.

The notions of contains, occurs, start- and end-point of a string w are also extended to $\mathcal{H}_n$. For w covered by $\mathcal{K}_n$, the start- and end-points of $\rho_n(w)$ are the start and end-points of w. Hence, the start-point of $\rho_n(w)$ is the integer occurring between the first \$ and # signs, and the notions of start- and end-points is only extended to images of strings covered by $\mathcal{K}_n$. The notion of containing an integer, on the other hand, is extended to every string v which is a covered by $\mathcal{H}_n$. That is, v *contains* any integer encoded by $\lceil \log n \rceil$ consecutive bits in v. Clearly, for any w covered by $\mathcal{K}_n$, $\rho_n(w)$ contains exactly the integers contained in w, but the notion is a bit more general. For instance, for $n = 4$, the string 0\$01#11\$10 contains 1, 2, and 3 but its start- and end-point are undefined as it is not the image of a string covered by $\mathcal{K}_n$.

For a regular expression r, we say that i is a *sidekick* of r when it occurs in every non-empty string defined by r. A regular expression s is a *starred subexpression* of a regular expression r when s is a subexpression of r and is of the form t^*. We say that a regular expression r is *proper* if every starred subexpression of r has a sidekick.

Lemma 10. Any starred subexpression s of a regular expression r defining $\mathcal{H}_n$ is proper.

Proof. We prove this lemma by a detailed examination of the structure of strings defined by s. We start by making the following observation. For any string $w \in L(s)$, there exist strings $u, u' \in \Sigma_{\mathcal{H}}^*$ such that $uwu' \in L(r)$. Furthermore, w can be pumped in uwu' and still form a string defined by r. That is, for every $j \in \mathbb{N}$, $uw^ju' \in L(r)$. In addition, for every other $w' \in L(s)$, $uw'u' \in L(r)$.

Let w be a non-empty string in $L(s)$ and let u, u' be such that $uwu' \in L(r)$. Then w must contain at least one \$-symbol. Towards a contradiction, suppose it does not. If w contains a # then $uwwu' \in L(r)$ but $uwwu' \notin \mathcal{H}_n$ which leads to the desired contradiction. If w contains no # and therefore only consists of 0's and 1's, then $uw^nu' \in L(r)$ but $uw^nu' \notin \mathcal{H}_n$ which again is a contradiction. In a similar way, one can show that (i) w contains at least one #-symbol; (ii) w contains an equal number of \$ and #-symbols; (iii) the \$ and #-symbols must alternate; and (iv) between any consecutive \$ and #-symbol there is a string of length $\lceil \log(n) \rceil$ containing only 0's and 1's.

From the above it follows that w matches one of the following expressions:

1. $\alpha_1 = (0+1)^*\$(0+1)^{\lceil \log n \rceil}\#\Sigma_{\mathcal{H}}^*$
2. $\alpha_2 = \Sigma_{\mathcal{H}}^*\#(0+1)^{\lceil \log n \rceil}\$(0+1)^*$.

We refer to the strings defined by α_1 and α_2 as strings of type-1 and type-2, respectively. We next show that all strings defined by s are either all of type-1 or all of type-2. Towards a contradiction assume there is a type-1 string w_1 and a type-2 string w_2 in $L(s)$. Then, $w_2w_1 \in L(s)$ and thus there exist $u, u' \in \Sigma_{\mathcal{H}}^*$ such that $uw_2w_1u' \in L(r)$. However, because of the concatenation of w_2 and w_1, there are two \$-symbols without an intermediate #-symbol and therefore $uw_2w_1u' \notin \mathcal{H}_n$.

Assume that all strings defined by s are of type-1. We next argue that the substring of length $\lceil \log n \rceil$, that is, the integer i, between the first \$ and #-symbol is the same for every $w \in L(s)$ which gives us our sidekick. For a type-1 string, we refer to this integer as the start block. Towards a contradiction, suppose that w, w_1 and w_2 are non-empty strings in $L(s)$ such that w_1 and w_2 have different start blocks. Let u, u' be such that $uww_1u' \in L(r)$ and therefore $uww_1u' \in \mathcal{H}_n$. Now, uw contains at least one \$ and #-symbol. Therefore, by definition of $\mathcal{H}_n$, the value of the start block of w_1 is uniquely determined by uw. That is, it must be equal to the integer preceding the last \$-symbol in uw. Now also $uww_2u' \in L(r)$ as s is a starred subexpression, but $uww_2u' \notin \mathcal{H}_n$ as w_2 has a different start block, which yields the desired contradiction.

The same kind of reasoning can be used to show that s has a sidekick when all defined strings are of type-2. □

If there is a greatest integer m for which r covers w^m, we call m the *index* of w in r and denote it by $I_w(r)$. In this case we say that r is *w-finite*. Otherwise, we say that r is *w-infinite*. The index of a regular expression can be used to give a lower bound on its size according to the following lemma.

Lemma 11 ([30]). [1] For any regular expression r and string w, if r is w-finite, then $I_w(r) < 2|r|$.

Now, we can state the most important property of $\mathcal{H}_n$.

Lemma 12. Let $n \geq 2$. For any $C \subseteq \{0, \dots, n-1\}$ of cardinality k and $i \in C$, there exists a string w covered by $\mathcal{H}_n$ with start- and end-point i and only containing integers in C, such that any proper regular expression r which covers w is of size at least 2^k.

Proof. The proof is by induction on k. For $k = 1$, $C = \{i\}$. Then, define $w = \text{enc}(i)\$\text{enc}(i)\#$, which satisfies all conditions and any expression covering w must definitely have size at least 2.

For the inductive step, let $C = \{j_1, \dots, j_k\}$ and $i \in C$. Define $C_\ell = C \setminus \{j_{(\ell \mod k)+1}\}$ and let w_ℓ be the string given by the induction hypothesis with respect to C_ℓ (of size $k-1$) and j_ℓ. Note that $j_\ell \in C_\ell$. Further, define $m = 2^{k+1}$ and set w equal to the string

$$\text{enc}(j_1)\$\text{enc}(i)\#w_1^m\text{enc}(j_2)\$\text{enc}(j_1)\#w_2^m\text{enc}(j_3)\$\text{enc}(j_2)\#\cdots w_k^m\text{enc}(i)\$\text{enc}(j_k)\#.$$

Then, w is covered by $\mathcal{H}_n$, has i as start and end-point and only contains integers in C. It only remains to show that any expression r which is proper and covers w is of size at least 2^k.

Fix such a regular expression r. If r is w_ℓ-finite for some $\ell \leq k$ then, $I_{w_\ell}(r_k) \geq m = 2^{k+1}$, by construction of w. By Lemma 11, $|r| \geq 2^k$ and we are done.

Therefore, assume that r is w_ℓ-infinite for every $\ell \leq k$. For every $\ell \leq k$, consider all subexpressions of r which are w_ℓ-infinite. We will see that all minimal elements in this set of subexpressions must be starred subexpressions. Here and in the following, we say that an expression is minimal with respect to a set simply when no other expression in the set is a subexpression. Indeed, a subexpression of the form a or ε can never be w_ℓ-infinite and a subexpression

[1] In fact, in [30] the length of an expression is defined as the number of Σ-symbols occurring in it. However, since our length measure also contains these Σ-symbols, this lemma still holds in our setting.

of the form r_1r_2 or $r_1 + r_2$ can only be w_ℓ-infinite if r_1 and/or r_2 are w_ℓ-infinite and is thus not minimal with respect to w_ℓ-infinity. Among these minimal starred subexpressions for w_ℓ, choose one and denote it by s_ℓ. Let $E = \{s_1, \ldots, s_k\}$. Note that since r is proper, all its subexpressions are also proper. As in addition each s_ℓ covers w_ℓ, by the induction hypothesis the size of each s_ℓ is at least 2^{k-1}. Now, choose from E some expression s_ℓ such that s_ℓ is minimal with respect to the other elements in E.

As r is proper and s_ℓ is a starred subexpression of r, there is an integer j such that every non-empty string in $L(s_\ell)$ contains j. As, by the induction hypothesis, w_ℓ only contains integers in C, s_ℓ is w_ℓ-infinite, and s_ℓ is chosen to be minimal with respect to w_ℓ-infinity, it follows that $j \in C = \{j_1, \ldots, j_k\}$ must hold. Let k' be such that $j = j_{k'}$. By definition of the inductively obtained strings $w_1, \ldots, w_k$, we have that for $p = k' - 1$ (if $k' \geq 2$, and $p = k$, otherwise), w_p does not contain j. Denote by s_p the starred subexpression from E which is w_p-infinite. In particular, as j is a sidekick of s_ℓ, and s_p defines strings not containing j (recall that it is w_p-infinite, and minimal in this respect), s_ℓ and s_p cannot be the same subexpression of r.

Now, there are three possibilities:

- s_ℓ and s_p are completely disjoint subexpressions of r. That is, they are both not a subexpression of one another. By induction they must both be of size 2^{k-1} and thus $|r| \geq 2^{k-1} + 2^{k-1} = 2^k$.

- s_p is a strict subexpression of s_ℓ. This is not possible since s_ℓ is chosen to be a minimal element from E.

- s_ℓ is a strict subexpression of s_p. We show that if we replace s_ℓ by ε in s_p, then s_p is still w_p-infinite. It then follows that s_p still covers w_p, and thus s_p without s_ℓ is of size at least 2^{k-1}. As $|s_\ell| \geq 2^{k-1}$ as well it follows that $|r| \geq 2^k$.

  To see that s_p without s_ℓ is still w_p-infinite, recall that any non-empty string defined by s_ℓ contains j and j does not occur in w_p. Therefore, a full iteration of s_ℓ can never contribute to the matching of any number of repetitions of w_p. Or, more specifically, no non-empty word $w \in L(s_\ell)$ can ever be a substring of a word w_p^i, for any i. So, s_p can only lose its w_p-infinity by this replacement if s_ℓ contains a subexpression which is itself w_p-infinite. However, this then also is a subexpression of s_p and s_p is chosen to be minimal with respect to w_p-infinity, a contradiction. We can only conclude that s_p without s_ℓ is still w_p-infinite. $\square$

Since by Lemma 10 any expression defining $\mathcal{H}_n$ is proper, Theorem 9(2) directly follows from Lemma 12 by choosing $i = 0$, $k = n$. This concludes the proof of Theorem 9(2).

We now extend the results on $\mathcal{H}_n$, which is defined over the four-letter alphabet $\Sigma_{\mathcal{H}} = \{0, 1, \$, \#\}$, to a binary alphabet. Thereto, let $\Sigma_2 = \{a, b\}$, and define ρ_2 as $\rho_2(0) = aa$, $\rho_2(1) = ab$, $\rho_2(\$) = ba$, and $\rho_2(\#) = bb$. Then, as usual, for $w = a_1 \cdots a_n \in \Sigma_{\mathcal{H}}^*$, $\rho_2(w) = \rho_2(a_1) \cdots \rho_2(a_n)$. Then, we can define $\mathcal{H}_n^2$ as follows.

Definition 13. For $n \in \mathbb{N}$, let $\mathcal{H}_n^2 = \{\rho_2(w) \mid w \in \mathcal{H}_n\}$.

We can now easily extend Theorem 9 to $\mathcal{H}_n^2$, thus showing that in a translation from a DFA to an RE an exponential blow-up can not be avoided, even when the alphabet is fixed and binary.

Theorem 14. For any $n \in \mathbb{N}$, with $n \geq 2$,

1. there is a DFA A_n of size $\mathcal{O}(n^2 \log n)$ defining $\mathcal{H}_n^2$; and,
2. any regular expression defining $\mathcal{H}_n^2$ is of size at least 2^n.

Proof. The proof of Theorem 9 carries over almost literally to the current setting. For (1), the automaton A_n can be constructed very similarly as the DFA A_n in Theorem 9. The only difference is that for every transition, we have to add an additional state to the automaton. For instance, if there is a transition from state q to q', reading a 0, we have to insert a new state, say q_0, with transitions from q to q_0, and q_0 to q', each reading an a. Further, if after applying this transformation, we obtain a state which has two outgoing transitions with the same label, we can simply merge the two states to which the transitions point. The latter happens, for instance, when a state had an outgoing 0 and 1 transition, as $\rho_2(0) = aa$ and $\rho_2(1) = ab$. The obtained DFA accepts $\mathcal{H}_n^2$ and is of size $\mathcal{O}(n^2 \log n)$.

For (2), the proof of Theorem 9(2) also carries over almost literally. In particular, all starred subexpressions of an expression defining $\mathcal{H}_n^2$ must still have a sidekick. And, in the last and crucial step of the proof, if we replace the starred subexpression s_ℓ by ε in s_p, then s_p is still w_p-infinite. Hence, it can be proved that any regular expression defining $\mathcal{H}_n^2$ is of size at least 2^n. $\square$

We conclude the section by noting that Waizenegger [107] already claimed a result similar to Theorem 9 using a different binary encoding of $\mathcal{K}_n$. Unfortunately, we believe that a more sophisticated encoding as presented here is necessary, and hence the proof in [107] to be incorrect, as we discuss in some detail next.

He takes an approach similar to ours and defines a binary encoding of $\mathcal{K}_n$ as follows. For $n \in \mathbb{N}$, Σ_n consists of n^2 symbols. If we list these, in some unspecified order, and associate a natural number from 0 to $n^2 - 1$ to each of them, we can encode $\mathcal{K}_n$ using $\log(n^2)$ bits. For instance, if $n = 2$, then

$\Sigma_n = \{a_{0,0}, a_{0,1}, a_{1,0}, a_{1,1}\}$ and we can encode these by the numbers $0, 1, 2, 3$, respectively. Now, a string in $\mathcal{K}_n$ is encoded by replacing every symbol by its binary encoding and additionally adding an a and b to the start and end of each symbol. For instance, $a_{0,0}a_{0,1}$ becomes $a00ba01b$, which gives a language, say $\mathcal{W}_n$, over the four-letter alphabet $\{0, 1, a, b\}$.

Then, Waizenegger shows that $\mathcal{W}_n$ can be described by an NFA of size $\mathcal{O}(k^2 \cdot \log k^2)$, but every regular expression defining it must be of size at least exponential in n. The latter is done by using the same proof techniques as in [30]. However, in this proof it is claimed that if r_n is an expression defining $\mathcal{W}_n$, then every string defined by a starred subexpression of r_n must start with an a. This statement is incorrect. For instance, if r_2 is an expression defining W_2, and we still use the same encoding as above, then $((a0(0ba0)^*0b) + \varepsilon)r_2$ still defines W_2 but does not have this property. Albeit small, the mistake has serious consequences. It follows from this claim that every starred subexpression has a sidekick (using our terminology), and the rest of the proof builds upon this fact. To see the importance, consider the language $\mathcal{H}'_n$ obtained from $\mathcal{K}_n$ like $\mathcal{H}_n$ but without swapping the indices of the symbols in $\mathcal{K}_n$. As illustrated in Remark 8, $\mathcal{H}'_n$ can be defined by a regular expression of size $\mathcal{O}(n \log(n))$. However, if we assume that every string defined by a starred subexpression starts with a #, we can reuse our proof for $\mathcal{H}_n$ and show that any regular expression defining $\mathcal{H}'_n$ must be of at least exponential size which is clearly false.

To summarize, Waizenegger's claim that the specific encoding of the n^2 symbols in Σ does not matter in the proof for Theorem 9 is incorrect. Our encoding used in defining $\mathcal{H}_n$ allows to prove the sidekick lemma (Lemma 10) in a correct way.

3.2 Complementing Regular Expressions

It is known that extended regular expressions are non-elementary more succinct than classical ones [27, 102]. Intuitively, each exponent in the tower requires nesting of an additional complement. In this section, we show that in defining the complement of a single regular expression, a double exponential size increase cannot be avoided in general. In contrast, when the expression is deterministic its complement can be computed in polynomial time.

Theorem 15. 1. For every regular expression r over Σ, a regular expression s with $L(s) = \Sigma^* \setminus L(r)$ can be constructed in time $2^{2^{\mathcal{O}(n)}}$.

2. Let Σ be a binary alphabet. For every $n \in \mathbb{N}$, there is a regular expression r_n of size $\mathcal{O}(n)$ such that any regular expression r defining $\Sigma^* \setminus L(r_n)$ is of size at least 2^{2^n}.

Proof. (1) Let $r \in \mathrm{RE}$. We first construct a DFA A, with $L(A) = \Sigma^* \setminus L(r)$ and then construct the regular expression s equivalent to A. According to Theorem 2(3) and (4) A contains at most $2^{|r|+1}$ states and can be constructed in time exponential in the size of r. Then, by Theorem 2(1), the total algorithm is in time $2^{2^{\mathcal{O}(n)}}$.
(2) Take Σ as $\Sigma_{\mathcal{H}}$, that is, $\{0, 1, \$, \#\}$. Let $n \in \mathbb{N}$. We define an expression r'_n of size $\mathcal{O}(n)$, such that $\Sigma^* \setminus L(r'_n) = \mathcal{H}_{2^n}$. By Theorem 9, any regular expression defining $\mathcal{H}_{2^n}$ is of size exponential in 2^n, that is, of size 2^{2^n}. This hence already proves that the theorem holds for languages over an alphabet of size four. Afterwards, we show that it also holds for alphabets of size two.

The expression r'_n is then defined as the disjunction of the following expressions:

- all strings that do not start with a prefix in $(0+1)^n\$0^n$:

$$\Sigma^{0,2n} + \Sigma^{0,n-1}(\$ + \#)\Sigma^* + \Sigma^n(0+1+\#)\Sigma^* + \Sigma^{n+1,2n}(1+\$+\#)\Sigma^*$$

- all strings that do not end with a suffix in $1^n\$(0+1)^n\#$:

$$\Sigma^{0,2n+1} + \Sigma^*(0+1+\$) + \Sigma^*(\$+\#)\Sigma^{1,n} + \\ \Sigma^*(0+1+\#)\Sigma^{n+1} + \Sigma^*(0+\$+\#)\Sigma^{n+2,2n+1}$$

- all strings where a \$ is not followed by a string in $(0+1)^n\#$:

$$\Sigma^*\$\big(\Sigma^{0,n-1}(\#+\$) + \Sigma^n(0+1+\$)\big)\Sigma^*$$

- all strings where a non-final # is not followed by a string in $(0+1)^n\$$:

$$\Sigma^*\#\big(\Sigma^{0,n-1}(\#+\$) + \Sigma^n(0+1+\#)\big)\Sigma^*$$

- all strings where the corresponding bits of corresponding blocks are different:

$$((0+1)^* + \Sigma^*\#(0+1)^*)0\Sigma^{3n+2}1\Sigma^* + ((0+1)^* + \Sigma^*\#(0+1)^*)1\Sigma^{3n+2}0\Sigma^*.$$

Notice here that in constructing each of the above regular expressions, we only need to consider words which are not yet defined by one of the foregoing expressions. For instance, the last expression above is only properly defined over words of the form $((0+1)^n\$(0+1)^n\#)^*$, as all other strings are already defined by the previous expressions. Then, it should be clear that a string over $\{0, 1, \$, \#\}$ is matched by none of the above expressions if and only if it belongs to $\mathcal{H}_{2^n}$. So, the complement of r'_n defines exactly $\mathcal{H}_{2^n}$.

We now extend this result to a binary alphabet $\Sigma = \{a, b\}$. Thereto, let $\rho_2(r'_n)$ be the expression obtained by replacing every symbol σ occurring in r'_n by $\rho_2(\sigma)$. Then, the expression $r_n = ((a+b)(a+b))^*(a+b) + \rho_2(r'_n)$ is the desired expression as its complement is exactly $\mathcal{H}^2_{2^n}$, which by Theorem 14 must be of size at least 2^{2^n}. To see that the complement of r_n is indeed $\mathcal{H}^2_{2^n}$, notice that the expression $((a+b)(a+b))^*(a+b)$ defines exactly all strings which are not the encoding of some string $w \in \Sigma^*_{\mathcal{H}}$. Then, $\rho_2(r'_n)$ captures exactly all other strings which do represent a string $w \in \Sigma^*_{\mathcal{H}}$, but are not in $\mathcal{H}^2_{2^n}$. Hence, the complement of r_n is exactly $\mathcal{H}^2_{2^n}$. $\square$

The previous theorem essentially shows that in complementing a regular expression, there is no better algorithm than translating to a DFA, computing the complement and translating back to a regular expression which includes two exponential steps. However, when the given regular expression is deterministic, a corresponding DFA can be computed in quadratic time through the Glushkov construction [17] eliminating already one exponential step. It should be noted that this construction, which we refer to as Glushkov construction, was introduced by [14], and often goes by the name position automata [57]. However, as in the context of deterministic expressions the name Glushkov construction is the most common, we will consistently use this naming.

The proof of the next theorem shows that the complement of the Glushkov automaton of a deterministic expression can directly be defined by a regular expression of polynomial size.

Theorem 16. For any deterministic regular expression r over an alphabet Σ, a regular expression s defining $\Sigma^* \setminus L(r)$ can be constructed in time $\mathcal{O}(n^3)$, where n is the size of r.

Proof. Let r be a deterministic expression over Σ and fix a marking $\overline{r}$ of r. We introduce some notation.

- The set not-first(r) contains all Σ-symbols which are not the first symbol in any word defined by r, that is, not-first$(r) = \Sigma \setminus \text{first}(r)$.
- For any symbol $\overline{x} \in \text{Char}(\overline{r})$, the set not-follow$(r, \overline{x})$ contains all Σ-symbols of which no marked version can follow $\overline{x}$ in any word defined by $\overline{r}$. That is, not-follow$(r, \overline{x}) = \Sigma \setminus \{\text{dm}(\overline{y}) \mid \overline{y} \in \text{Char}(\overline{r}) \wedge \exists \overline{w}, \overline{w}' \in \text{Char}(\overline{r})^*, \overline{wxyw}' \in L(\overline{r})\}$. [2]
- Recall that last$(\overline{r})$ contains all *marked* symbols which are the last symbol of some word defined by $\overline{r}$.

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

We define the following regular expressions:

- $\text{init}(r) = \begin{cases} \text{not-first}(r)\Sigma^* & \text{if } \varepsilon \in L(r); \text{ and} \\ \varepsilon + \text{not-first}(r)\Sigma^* & \text{if } \varepsilon \notin L(r). \end{cases}$
- For every $\overline{x} \in \text{Char}(\overline{r})$, $\overline{r}_{\overline{x}}$ will denote an expression defining $\{\overline{wx} \mid \overline{w} \in \text{Char}(\overline{r})^* \wedge \exists \overline{u} \in \text{Char}(\overline{r})^*, \overline{wxu} \in L(\overline{r})\}$. That is, all prefixes of strings in $\overline{r}$ ending in $\overline{x}$. Then, let $r_{\overline{x}}$ be $\text{dm}(\overline{r}_{\overline{x}})$.

We are now ready to define s:

$$\text{init}(r) + \bigcup_{\overline{x} \notin \text{last}(\overline{r})} r_{\overline{x}}(\varepsilon + \text{not-follow}(r, \overline{x})\Sigma^*) + \bigcup_{\overline{x} \in \text{last}(\overline{r})} r_{\overline{x}}\text{not-follow}(r, \overline{x})\Sigma^*.$$

We conclude by showing that s can be constructed in time cubic in the size of r and that s defines the complement of r. We prove that $L(s) = \Sigma^* \setminus L(r)$ by highlighting the correspondence between s and the complement of the Glushkov automaton G_r of r. The Glushkov automaton G_r is the DFA (Q, q_0, δ, F), where

- $Q = \{q_0\} \cup \{q_{\overline{x}} \mid \overline{x} \in \text{Char}(\overline{r})\}$;
- $F = \{q_{\overline{x}} \mid \overline{x} \in \text{last}(\overline{r})\}$ (plus q_0 if $\varepsilon \in L(r)$); and
- for $\overline{x}, \overline{y} \in \text{Char}(\overline{r})$, there is
  - a transition $(q_0, \text{dm}(\overline{x}), q_{\overline{x}}) \in \delta$, if $\overline{x} \in \text{first}(\overline{r})$; and,
  - a transition $(q_{\overline{x}}, \text{dm}(\overline{y}), q_{\overline{y}}) \in \delta$, if $\overline{y}$ follows $\overline{x}$ in some word defined by $\overline{r}$.

It is known that $L(r) = L(G_r)$ and that G_r is deterministic whenever r is deterministic [17].

The complement automaton $G_r^c = (Q^c, q_0, \delta^c, F^c)$ is obtained from G_r by making it complete and interchanging final and non-final states. Formally, $Q^c = Q \cup \{q_-\}$ (with $q_- \notin Q$) and δ^c contains δ plus the triples (q_-, a, q_-), for every $a \in \Sigma$, and (q, a, q_-) for every state $q \in Q$ and symbol $a \in \Sigma$ for which there is no $q' \in Q$ with $(q, a, q') \in \delta$. Finally, $F^c = \{q_-\} \cup (Q \setminus F)$. Clearly, $L(G_r^c) = \Sigma^* \setminus L(G_r)$.

Now, we show that $L(s) = L(G_r^c)$. First, by definition of s, $\varepsilon \in L(s)$ if and only if $\varepsilon \notin L(r)$ if and only if $\varepsilon \in L(G_r^c)$. We prove that for any non-empty word w, $w \in L(s)$ if and only if $w \in L(G_r^c)$, from which the lemma follows. Thereto, we show that the non-empty words defined by the different disjuncts of s correspond exactly to subsets of the language G_r^c.

- $\text{init}(r)$ defines exactly the non-empty words for which G_r^c immediately goes from q_0 to $q_\_$ and reads the rest of the word while in $q_\_$.

- For any $\overline{x} \in \text{last}(\overline{r})$, $r_{\overline{x}}(\text{not-follow}(r, \overline{x})\Sigma^*)$ defines exactly all strings w for which G_r^c arrives in $q_{\overline{x}}$ after reading a part of w, goes to $q_\_$ and reads the rest of w there.

- For any $\overline{x} \notin \text{last}(r)$, $r_{\overline{x}}(\varepsilon + \text{not-follow}(r, \overline{x})\Sigma^*)$ defines exactly all strings w for which G_r^c either (1) arrives in $q_{\overline{x}}$ after reading w and accepts because $q_{\overline{x}}$ is an accepting state; or (2) arrives in $q_{\overline{x}}$ after reading a part of w, goes to $q_\_$ and reads the rest of w there.

Note that because there are no incoming transitions in q_0, and $q_\_$ only has transitions to itself, we have described exactly all accepting runs of G_r^c. Therefore, any non-empty string $w \in L(s)$ if and only if $w \in G_r^c$.

We now show that s can be computed in time cubic in the size of r. By a result of Brüggemann-Klein [15] the Glushkov automaton G_r corresponding to r as defined above can be computed in time quadratic in the size of r. Using G_r, the sets $\text{not-first}(r)$, $\text{not-follow}(r, \overline{x})$ and $\text{last}(r)$ can be computed in time quadratic in the size of r. So, all sets can be computed in time cubic in the size of r. The expression $\text{init}(r)$ can be constructed in linear time. We next show that for any $\overline{x} \in \text{Char}(\overline{r})$, the expression $\overline{r}_{\overline{x}}$ can be constructed in time quadratic in the size of r. As $r_{\overline{x}} = \text{dm}(\overline{r}_x)$, it follows that s can be constructed in cubic time. The expression $\overline{r}_{\overline{x}}$ is inductively defined as follows:

- For $\overline{r} = \varepsilon$ or $\overline{r} = \emptyset$, $\overline{r}_{\overline{x}} = \emptyset$.

- For $\overline{r} = \overline{y} \in \text{Char}(\overline{r})$,

$$\overline{r}_{\overline{x}} = \begin{cases} \overline{x} & \text{if } \overline{y} = \overline{x} \\ \emptyset & \text{otherwise.} \end{cases}$$

- For $\overline{r} = \alpha\beta$,

$$\overline{r}_{\overline{x}} = \begin{cases} \alpha_{\overline{x}} & \text{if } \overline{x} \text{ occurs in } \alpha \\ \alpha\beta_{\overline{x}} & \text{otherwise.} \end{cases}$$

- For $\overline{r} = \alpha + \beta$,

$$\overline{r}_{\overline{x}} = \begin{cases} \alpha_{\overline{x}} & \text{if } \overline{x} \text{ occurs in } \alpha \\ \beta_{\overline{x}} & \text{otherwise.} \end{cases}$$

- For $\overline{r} = \alpha^*$, $\overline{r}_{\overline{x}} = \alpha^*\alpha_{\overline{x}}$

The correctness is easily proved by induction on the structure of $\overline{r}$. Note that there are no ambiguities in the inductive definition of concatenation and disjunction since the expressions are marked and therefore every marked symbol occurs only once in the expression. For the time complexity, notice that all steps in the above inductive definition, except for the case $\overline{r} = \alpha^*$, are linear. Further, for $\overline{r} = \alpha^*$, $\overline{r}_{\overline{x}} = \alpha^*\alpha_{\overline{x}}$, and hence α is doubled. However, as the inductive construction continues on only one of the two operands it follows that the complete construction is at most quadratic. □

We illustrate the construction in the previous proof by means of an example. Let $r = a(ab^*c)^*$, and $\overline{r} = a_1(a_2b_3^*c_4)^*$. Then,

- $\overline{r}_{a_1} = a_1$,
- $\overline{r}_{a_2} = a_1(a_2b_3^*c_4)^*a_2$,
- $\overline{r}_{b_3} = a_1(a_2b_3^*c_4)^*a_2b_3^*b_3$,
- $\overline{r}_{c_4} = a_1(a_2b_3^*c_4)^*a_2b_3^*c_4$, and
- $\text{init}(r) = \varepsilon + (b+c)\Sigma^*$.

Then the complement of r is defined by

$$\begin{aligned}&\varepsilon + (b+c)\Sigma^* \\ &\qquad + a(ab^*c)^*a(\varepsilon + a\Sigma^*) + a(ab^*c)^*ab^*b(\varepsilon + a\Sigma^*) \\ &\qquad\qquad + a((b+c)\Sigma^*) + a(ab^*c)^*ab^*c((b+c)\Sigma^*).\end{aligned}$$

We conclude this section by remarking that deterministic regular expressions are not closed under complement and that the constructed expression s is therefore not necessarily deterministic.

3.3 Intersecting Regular Expressions

In this section, we study the succinctness of intersection. In particular, we show that the intersection of two (or any fixed number) and an arbitrary number of regular expressions are exponentially and double exponentially more succinct than regular expressions, respectively. Actually, the exponential bound for a fixed number of expressions already holds for single-occurrence regular expressions [3], whereas the double exponential bound for an arbitrary

[3]Recall that an expression is single-occurrence if every alphabet symbol occurs at most once in it.

number of expressions only carries over to deterministic expressions. For single-occurrence expressions this can again be done in exponential time.

In this respect, we introduce a slightly altered version of $\mathcal{H}_n$.

Definition 17. Let $\Sigma_{\mathcal{D}} = \{0, 1, \$, \#, \triangle\}$. For all $n \in \mathbb{N}$, $\mathcal{D}_n = \{\rho_n(w)\triangle \mid \mathcal{K}_n$ covers $w \wedge |w|$ is even$\}$.

Here, we require that $\mathcal{K}_n$ covers w, instead of requiring $w \in \mathcal{K}_n$, to allow also for strings with a different start- and end-point than 0 and $n-1$, respectively. This will prove technically convenient later on, but does not change the structure of the language.

We also define a variant of $\mathcal{K}_n$ which only slightly alters the $a_{i,j}$ symbols in $\mathcal{K}_n$. Thereto, let $\Sigma_n^\circ = \{a_{i^\circ,j}, a_{i,j^\circ} \mid 0 \leq i,j < n\}$, $\hat{\rho}(a_{i,j}a_{j,k}) = \rhd_i a_{i,j^\circ} a_{j^\circ,k}$ and $\hat{\rho}(a_{i_0,i_1}a_{i_1,i_2}\cdots a_{i_{k-2},i_{k-1}}a_{i_{k-1},i_k}) = \hat{\rho}(a_{i_0,i_1}a_{i_1,i_2})\cdots\hat{\rho}(a_{i_{k-2},i_{k-1}}a_{i_{k-1},i_k})$, where k is even.

Definition 18. Let $n \in \mathbb{N}$ and $\Sigma_{\mathcal{E}}^n = \Sigma_n^\circ \cup \{\rhd_0, \triangle_0, \ldots, \rhd_{n-1}, \triangle_{n-2}\}$. Then, $\mathcal{E}_n = \{\hat{\rho}(w)\triangle_i \mid \mathcal{K}_n$ covers $w \wedge |w|$ is even $\wedge\ i$ is the end-point of $w\} \cup \{\triangle_i \mid 0 \leq i < n\}$.

Note that words in $\mathcal{E}_n$ are those in $\mathcal{K}_n$ where every odd position is promoted to a circled one ($^\circ$), and triangles labeled with the non-circled positions are added. For instance, the string $a_{2,4}a_{4,3}a_{3,3}a_{3,0}$ which is covered by $\mathcal{K}_5$ is mapped to the string $\rhd_2 a_{2,4^\circ}a_{4^\circ,3} \rhd_3 a_{3,3^\circ}a_{3^\circ,0}\triangle_0 \in \mathcal{E}_5$.

We make use of the following property:

Lemma 19. Let $n \in \mathbb{N}$.

1. Any regular expression defining $\mathcal{D}_n$ is of size at least 2^n.

2. Any regular expression defining $\mathcal{E}_n$ is of size at least 2^{n-1}.

Proof. (1) Analogous to Lemma 10, any regular expression r defining $\mathcal{D}_n$ must also be proper and must furthermore cover any string $w \in \mathcal{H}_n$. By choosing $i = 0$ and $k = n$ in Lemma 12, we see that r must be of size at least 2^n.

(2) Let $n \in \mathbb{N}$ and $r_{\mathcal{E}}$ be a regular expression defining $\mathcal{E}_n$. Let $r^2_{\mathcal{K}}$ be the regular expression obtained from $r_{\mathcal{E}}$ by replacing $\rhd_i$ and $\triangle_i$, with $i < n$, by ε and any $a_{i^\circ,j}$ or a_{i,j°, with $0 \leq i,j < n$, by $a_{i,j}$. Then, $|r^2_{\mathcal{K}}| \leq |r_{\mathcal{E}}|$ and $r^2_{\mathcal{K}}$ defines exactly all strings of even length in $\mathcal{K}_n$ plus ε, and thus also covers every string in $\mathcal{K}_n$. Since the proof in [30] also constructs a string w covered by $\mathcal{K}_n$ such that any proper expression covering w must be of size at least 2^{n-1}, it immediately follows that $r^2_{\mathcal{K}}$ and thus $r_{\mathcal{E}}$ must be of size at least 2^{n-1}. □

The next theorem shows the succinctness of the intersection operator.

Theorem 20. 1. For any $k \in \mathbb{N}$ and regular expressions $r_1, \dots, r_k$, a regular expression defining $\bigcap_{i \leq k} L(r_k)$ can be constructed in time $2^{\mathcal{O}((m+1)^k)}$, where $m = \max\{|r_i| \mid 1 \leq i \leq k\}$.

2. For every $n \in \mathbb{N}$, there are SOREs r_n and s_n of size $\mathcal{O}(n^2)$ such that any regular expression defining $L(r_n) \cap L(s_n)$ is of size at least 2^{n-1}.

3. For every $n \in \mathbb{N}$, there are deterministic regular expressions $r_1, \dots, r_m$, with $m = 2n + 1$, of size $\mathcal{O}(n)$ such that any regular expression defining $\bigcap_{i \leq m} L(r_i)$ is of size at least 2^{2^n}.

4. Let $r_1, \dots, r_n$ be SOREs. A regular expression defining $\bigcap_{i \leq n} L(r_n)$ can be constructed in time $2^{\mathcal{O}(m)}$, where $m = \sum_{i \leq n} |r_i|$.

Proof. (1) First, construct NFAs $A_1, \dots, A_k$ such that $L(A_i) = L(r_i)$, for any $i \leq k$. If $m = \max\{|r_i| \mid 1 \leq i \leq k\}$, then by Theorem 2(4) any A_i has at most $m + 1$ states and can be constructed in time $\mathcal{O}(m \cdot |\Sigma|)$. Then, an NFA A with $(m+1)^k$ states, such that $L(A) = \bigcap_{i \leq k} L(A_i)$, can be constructed in time $\mathcal{O}((m+1)^k)$ by means of a product construction. By Theorem 2(1), a regular expression defining $L(A)$ can then be constructed in time $2^{\mathcal{O}((m+1)^k)}$.

(2) Let $n \in \mathbb{N}$. By Lemma 19(2), any regular expression defining M_n is of size at least 2^{n-1}. We define SOREs r_n and s_n of size quadratic in n, such that $L(r_n) \cap L(s_n) = \mathcal{E}_n$. We start by partitioning $\Sigma^n_{\mathcal{E}}$ in two different ways. To this end, for every $i < n$, define $\text{Out}_i = \{a_{i,j^\circ} \mid 0 \leq j < n\}$, $\text{In}_i = \{a_{j^\circ,i} \mid 0 \leq j < n\}$, $\text{Out}_{i^\circ} = \{a_{i^\circ,j} \mid 0 \leq j < n\}$, and, $\text{In}_{i^\circ} = \{a_{j,i^\circ} \mid 0 \leq j < n\}$. Then,

$$\Sigma^n_{\mathcal{E}} = \bigcup_i \text{In}_i \cup \text{Out}_i \cup \{\triangleright_i, \triangle_i\} = \bigcup_{i^\circ} \text{In}_{i^\circ} \cup \text{Out}_{i^\circ} \cup \{\triangleright_i, \triangle_i\}.$$

Further, define

$$r_n = \Big((\triangleright_0 + \cdots + \triangleright_{n-1}) \bigcup_{i^\circ} \text{In}_{i^\circ}\text{Out}_{i^\circ}\Big)^* (\triangle_0 + \cdots + \triangle_{n-1})$$

and

$$s_n = \Big(\bigcup_i (\text{In}_i + \varepsilon)(\triangleright_i + \triangle_i)(\text{Out}_i + \varepsilon)\Big)^*.$$

Now, r_n checks that every string consists of a sequence of blocks of the form $\triangleright_i a_{j,k^\circ} a_{k^\circ,\ell}$, for $i, j, k, \ell < n$, ending with a $\triangle_i$, for $i < n$. It thus sets the format of the strings and checks whether the circled indices are equal. Further, s_n checks whether the non-circled indices are equal and whether the triangles have the correct indices. Since the alphabet of $\mathcal{E}_n$ is of size $\mathcal{O}(n^2)$, also r_n and s_n are of size $\mathcal{O}(n^2)$.

(3) Let $n \in \mathbb{N}$. We define $m = 2n + 1$ deterministic regular expressions of size $\mathcal{O}(n)$, such that their intersection defines $\mathcal{D}_{2^n}$. By Lemma 19(1), any regular expression defining $\mathcal{D}_{2^n}$ is of size at least 2^{2^n} and the theorem follows. For ease of readability, we denote $\Sigma_{\mathcal{D}}$ simply by Σ. The expressions are as follows. There should be an even length sequence of blocks:

$$\big((0+1)^n\$(0+1)^n\#(0+1)^n\$(0+1)^n\#\big)^*\triangle.$$

For all $i \in \{0, \dots, n-1\}$, the $(i+1)$th bit of the two numbers surrounding an odd $\#$ should be equal:

$$\big(\Sigma^i(0\Sigma^{3n+2}0 + 1\Sigma^{3n+2}1)\Sigma^{n-i-1}\#\big)^*\triangle.$$

For all $i \in \{0, \dots, n-1\}$, the $(i+1)$th bit of the two numbers surrounding an even $\#$ should be equal:

$$\Sigma^{2n+2}\Big(\Sigma^i(0\Sigma^{2n-i+1}(\triangle + \Sigma^{n+i+1}0\Sigma^{n-i-1}\#) + \\ (1\Sigma^{2n-i+1}(\triangle + \Sigma^{n+i+1}1\Sigma^{n-i-1}\#)))\Big)^*.$$

Clearly, the intersection of the above expressions defines $\mathcal{D}_{2^n}$. Furthermore, every expression is of size $\mathcal{O}(n)$ and is deterministic as the Glushkov construction translates them into a DFA [17].

(4) We show that given SOREs $r_1, \dots, r_n$, we can construct an NFA A with $|\Sigma| + 1$ states defining $\bigcap_{i\leq n} L(r_i)$ in time cubic in the sizes of $r_1, \dots, r_n$. It then follows from Theorem 2(1) that an expression defining $\bigcap_{i\leq n} L(r_i)$ can be constructed in time $2^{\mathcal{O}(m)}$, where $m = \sum_{i\leq n} |r_i|$ since $m \geq |\Sigma|$.

We first construct NFAs $A_1, \dots, A_n$ such that $L(A_i) = L(r_i)$, for any $i \leq n$, by using the Glushkov construction [15]. This construction creates an automaton which has an initial state, with only outgoing transitions, and additionally one state for each symbol in the regular expressions. Furthermore, all incoming transitions for that state are labeled with that symbol. So, we could also say that each state is labeled with a symbol, and that all incoming transitions carry the label of that state. Since $r_1, \dots, r_n$ are SOREs, for every symbol there exists at most one state labeled with that symbol in any A_i. Now, let $A_i = (Q_i, q_0^i, \delta_i, F_i)$ then we say that $Q_i = \{q_0^i\} \cup \{q_a^i \mid a \in \Sigma\}$, where q_a^i is the state labeled with a. For ease of exposition, if a symbol a does not occur in an expression r_i, we add a state q_a^i to Q_i which does not have any incoming or outgoing transitions.

Now, we are ready to construct the NFA $A = (Q, q_0, \delta, F)$ defining the intersection of $A_1, \dots, A_n$. First, Q has again an initial state and one state for each symbol: $Q = \{q_0\} \cup \{q_a \mid a \in \Sigma\}$. A state is accepting if all its corresponding states are accepting: $F = \{q_a \mid \forall i \leq n, q_a^i \in F_i\}$. Here, a can denote 0 or an alphabet symbol. Finally, there is a transition between q_a and q_b if there is a transition between q_a^i and q_b^i, in every A_i: $\delta = \{(q_a, b, q_b) \mid \forall i \leq n, (q_a^i, b, q_b^i) \in \delta^i\}$. Now, $L(A) = \bigcap_{i \leq n} L(A_i)$. Since the Glushkov construction takes quadratic time [15], and we have to construct n automata, the total construction can be done in cubic time. □

We note that the lower bounds in Theorem 20(2) and (3) do not make use of a fixed size alphabet. Notice that for the results in Theorem 20(2) it is impossible to avoid this, as the number of SOREs over an alphabet of fixed size is finite. However, if we weaken the statements and allow to use unrestricted regular expressions in Theorem 20(2) and (3) (instead of SOREs and deterministic expressions) it is easy to adapt the proofs to use the language $\mathcal{K}_n^2$, and hence only use a binary alphabet.

We can thus obtain the following corollary.

Corollary 21. Let Σ be an alphabet with $|\Sigma| = 2$:

1. For every $n \in \mathbb{N}$, there are regular expressions r_n and s_n over Σ, of size $\mathcal{O}(n^2)$, such that any regular expression defining $L(r_n) \cap L(s_n)$ is of size at least 2^{n-1}.

2. For every $n \in \mathbb{N}$, there are regular expressions $r_1, \ldots, r_m$ over Σ, with $m = 2n + 1$, of size $\mathcal{O}(n)$, such that any regular expression defining $\bigcap_{i \leq m} L(r_i)$ is of size at least 2^{2^n}.

4

Succinctness of Extended Regular Expressions

Next to XML schema languages, regular expressions are used in many applications such as text processors and programming languages [108]. These applications, however, usually do not restrict themselves to the standard regular expression using disjunction (+), concatenation (·) and star (*), but also allow the use of additional operators. Although these operators mostly do not increase the expressive power of the regular expressions, they can have a drastic impact on succinctness, thus making them harder to handle. For instance, it is well-known that expressions extended with the complement operator can describe certain languages non-elementary more succinct than standard regular expressions or finite automata [102].

In this chapter, we study the succinctness of regular expressions extended with counting (RE(#)), intersection (RE($\cap$)), and interleaving (RE(&)) operators. The counting operator allows for expressions such as $a^{2,5}$, specifying that there must occur at least two and at most five a's. These RE(#)s are used in egrep [58] and Perl [108] patterns and in the XML schema language XML Schema [100]. The class RE($\cap$) is a well studied extension of the regular expressions, and is often referred to as the semi-extended regular expressions. The interleaving operator allows for expressions such as a & b & c, specifying that a, b, and c may occur in any order, and is used, for instance, in the XML schema language Relax NG [22].

A problem we consider, is the translation of extended regular expressions into (standard) regular expressions. For RE(#) the complexity of this translation is exponential [64] while it follows from the results in the previous chapter that for RE($\cap$) it is double exponential.[1] We show that also in constructing an expression for the interleaving of a set of expressions (an hence also for an RE(&)) a double exponential size increase can not be avoided. This is the main technical result of the chapter. Apart from a pure mathematical interest, the latter result has two important consequences. First, it prohibits an efficient translation from Relax NG (which allows interleaving) to XML Schema Definitions (which only allows a very limited form of interleaving). However, as XML Schema is the widespread W3C standard, and Relax NG is a more flexible alternative, such a translation would be more than desirable. A second consequence concerns the automatic discovery of regular expressions describing a set of given strings. The latter problem occurs in the learning of XML schema languages [8, 9, 11]. At present these algorithms do not take into account the interleaving operator, but for Relax NG this would be wise as this would allow to learn significantly smaller expressions.

It should be noted here that Gruber and Holzer [51] obtained similar results. They show that any regular expression defining the language $(a_1b_1)^* \mathbin{\&} \cdots \mathbin{\&} (a_nb_n)^*$ must be of size at least double exponential in n. Compared to the result in this chapter, this gives a tighter bound ($2^{2^{\Omega(n)}}$ instead of $2^{2^{\Omega(\sqrt{n})}}$), and shows that the double exponential size increase already occurs for very simple expressions. On the other hand, the alphabet of the counterexamples grows linear with n, whereas the alphabet size is constant for the languages in this chapter. Over a constant size alphabet, they improved our $2^{2^{\Omega(\sqrt{n})}}$ bound to $2^{2^{\Omega(n/\log n)}}$.

We also consider the translation of extended regular expressions to NFAs. For the standard regular expressions, it is well-known that such a translation can be done efficiently [15]. Therefore, when considering problems such as membership, equivalence, and inclusion testing for regular expressions the first step is almost invariantly a translation to a finite automaton. For extended regular expressions, such an approach is less fruitful. We show that an RE($\&, \cap, \#$) can be translated in exponential time into an NFA. However, it has already been shown by Kilpelainen and Tuhkanen [64] and Mayer and Stockmeyer [79] that such an exponential size increase can not be avoided for RE(#) and RE(&), respectively. For the translation from RE($\cap$) to NFAs, a $2^{\Omega(\sqrt{n})}$ lower bound is reported in [90], which we here improve to $2^{\Omega(n)}$.

As the translation of extended regular expressions to NFAs already involves an exponential size increase, it is natural to ask what the size increase for DFAs

[1]The bound following from the results in the previous chapter is $2^{2^{\Omega(\sqrt{n})}}$. This has been improved in [51] to $2^{2^{\Omega(n)}}$.

	NFA	DFA	RE
RE(#)	$2^{\Omega(n)}$ [64]	$2^{2^{\Omega(n)}}$ (Prop. 28)	$2^{\theta(n)}$ [64]
RE(∩)	$2^{\Omega(n)}$ (Prop. 26)	$2^{2^{\Omega(n)}}$ (Thm. 29)	$2^{2^{\theta(n)}}$ [51]
RE(&)	$2^{\Omega(n)}$ [79]	$2^{2^{\Omega(\sqrt{n})}}$ (Thm. 31)	$2^{2^{\Omega(\sqrt{n})}}$ (Thm. 39)
RE(&, ∩, #)	$2^{\mathcal{O}(n)}$ (Prop. 25)	$2^{2^{\mathcal{O}(n)}}$ (Prop. 27)	$2^{2^{\mathcal{O}(n)}}$ (Prop. 32)

(a)

	RE
RE ∩ RE	$2^{\Omega(n)}$ [48]
$\bigcap$ RE	$2^{2^{\Omega(\sqrt{n})}}$ (Ch. 3)
RE & RE	$2^{\Omega(n)}$ [48]

(b)

Figure 4.1: Table (a) gives an overview of the results in this chapter concerning the complexity of translating extended regular expressions into NFAs, DFAs, and regular expressions. Proposition and theorem numbers are given in brackets. Table (b) lists some related results obtained in the previous chapter and [48].

is. Of course, we can translate any NFA into a DFA in exponential time, thus giving a double exponential translation, but can we do better? For instance, from the results in the previous chapter, we can conclude that given a set of regular expressions, constructing an NFA for their intersection can not avoid an exponential size increase. However, it is not too hard to see that also a DFA of exponential size accepting their intersection can be constructed. In the present chapter, we show that this is not possible for the classes RE(#), RE(∩), and RE(&). For each class we show that in a translation to a DFA, a double exponential size increase can not be avoided. An overview of all results is given in Figure 4.1(a).

Related work. The different classes of regular expressions considered here have been well studied. In particular, the RE(∩) and its membership [90, 61, 71] and equivalence and emptiness [35, 89, 94] problems, but also the classes RE(#) [64, 83] and RE(&) [79] have received interest. Succinctness of regular expressions has been studied by Ehrenfeucht and Zeiger [30] and, more recently, by Ellul et. al [31], Gruber and Holzer [48, 50, 49], and Gruber and Johannsen [53]. Some relevant results are listed in Figure 4.1(b). Schott and Spehner give lower bounds for the translation of the interleaving of words to DFAs [96]. Also related, but different in nature, are the results on state complexity [111], in which the impact of the application of different operations on finite automata is studied.

Outline. In **Section 4.1** we give some additional necessary definitions and present some basic results. In **Sections 4.2**, **4.3**, and **4.4** we study the translation of extended regular expressions to NFAs, DFAs, and regular expressions, respectively.

4.1 Definitions and Basic Results

In this section we present some additional definitions and basic results used in the remainder of this chapter.

Intuitively, the *star height* of a regular expression r, denoted by $\mathrm{sh}(r)$, equals the number of nested stars in r. Formally, $\mathrm{sh}(\emptyset) = \mathrm{sh}(\varepsilon) = \mathrm{sh}(a) = 0$, for $a \in \Sigma$, $\mathrm{sh}(r_1 r_2) = sh(r_1{+}r_2) = \max\{\mathrm{sh}(r_1), \mathrm{sh}(r_2)\}$, and $\mathrm{sh}(r^*) = \mathrm{sh}(r){+}1$. The star height of a regular language L, denoted by $\mathrm{sh}(L)$, is the minimal star height among all regular expressions defining L.

The latter two concepts are related through the following theorem due to Gruber and Holzer [48], which will allow us to reduce our questions about the size of regular expressions to questions about the star height of regular languages.

Theorem 22 ([48])**.** Let L be a regular language. Then any regular expression defining L is of size at least $2^{\frac{1}{3}(\mathrm{sh}(L)-1)} - 1$.

A language L is *bideterministic* if there exists a DFA A, accepting L, such that the inverse of A is again deterministic. That is, A may have at most one final state and the automaton obtained by inverting every transition in A, and exchanging the initial and final state, is again deterministic.

A (directed) *graph* G is a tuple (V, E), where V is the set of *vertices* and $E \subseteq V \times V$ is the set of *edges*. A graph (U, F) is a *subgraph* of G if $U \subseteq V$ and $F \subseteq E$. For a set of vertices $U \subseteq V$, the *subgraph of G induced by U*, denoted by $G[U]$, is the graph (U, F), where $F = \{(u, v) \mid u, v \in U \wedge (u, v) \in E\}$.

A graph $G = (V, E)$ is *strongly connected* if for every pair of vertices $u, v \in V$, both u is reachable from v, and v is reachable from u. A set of edges $V' \subseteq V$ is a *strongly connected component (SCC)* of G if $G[V']$ is strongly connected and for every set V'', with $V' \subsetneq V''$, $G[V'']$ is not strongly connected. Let $\mathrm{SCC}(G)$ denote the set of strongly connected components of G.

We now introduce the *cycle rank* of a graph $G = (V, E)$, denoted by $\mathrm{cr}(G)$, which is a measure for the structural complexity of G. It is inductively defined as follows: (1) if G is acyclic or empty, then $\mathrm{cr}(G) = 0$, otherwise (2) if G is strongly connected, then $\mathrm{cr}(G) = \min_{v \in V} \mathrm{cr}(G[V \setminus \{v\}]) + 1$, and otherwise (3) $\mathrm{cr}(G) = \max_{V' \in \mathrm{SCC}(G)} \mathrm{cr}(G[V'])$.

We say that a graph H is a *minor* of a graph G if H can be obtained from G by removing edges, removing vertices, and contracting edges. Here, contracting an edge between two vertices u and v, means replacing u and v by a new vertex, which inherits all incoming and outgoing edges of u and v. Now, it has been shown by Cohen [23] that removing edges or nodes from a graph does not increase the cycle rank of a graph, while McNaughton [80] essentially showed that the same holds when contracting edges.

Lemma 23 ([23, 80])**.** If a graph H is a minor of a graph G, then $\mathrm{cr}(H) \leq \mathrm{cr}(G)$.

Let $A = (Q, q_0, \delta, F)$ be an NFA. The *underlying graph* G of A is the graph obtained by removing the labels from the transition edges of A, or more formally $G = (Q, E)$, with $E = \{(q, q') \mid \exists a \in \Sigma, (q, a, q') \in \delta\}$. In the following, we often abuse notation and for instance say the cycle rank of A, referring to the cycle rank of its underlying graph.

There is a strong connection between the star height of a regular language, and the cycle rank of the NFAs accepting it, as witnessed by the following theorem. Theorem 24(1) is known as Eggan's Theorem [29] and proved in its present form by Cohen [23]. Theorem 24(3) is due to McNaughton [80].

Theorem 24. For any regular language L,

1. $\mathrm{sh}(L) = \min\{\mathrm{cr}(A) \mid A$ is an NFA accepting L$\}$ [29, 23].
2. $\mathrm{sh}(L) \cdot |\Sigma| \geq \min\{\mathrm{cr}(A) \mid A$ is a non-returning state-labeled NFA such that $L = L(A)\}$.
3. if L is bideterministic, then $\mathrm{sh}(L) = \mathrm{cr}(A)$, where A is the minimal trimmed DFA accepting L. [80]

Proof. We only have to prove (2). Thereto, let L be a regular language over an alphabet Σ. By Eggan's Theorem (Theorem 24(1)), we know that there exists an NFA A, with $L(A) = L$ and $\mathrm{cr}(A) = \mathrm{sh}(L)$. We show that, given A, we can construct a non-returning state-labeled NFA B^{sl} equivalent to A such that $\mathrm{cr}(A) \cdot |\Sigma| \geq \mathrm{cr}(B^{\mathrm{sl}})$ from which the theorem follows.

Let $A = (Q, q_0, \delta, F)$ be an NFA over Σ, we construct B^{sl} in two steps. First, we construct a non-returning NFA $B = (Q^B, q_B, \delta^B, F^B)$, with $L(B) = L(A)$, as follows: $Q^B = Q \uplus \{q_B\}$, $\delta^B = \delta \cup \{(q_B, a, q) \mid q \in Q, a \in \Sigma, (q_0, a, q) \in \delta\}$, and $F^B = F$ if $q_0 \notin F$, and $F^B = F \cup \{q_B\}$, otherwise. Intuitively, B is A extended with a new initial state which only inherits the outgoing transitions of the old initial state. It should be clear that B is non-returning and $L(B) = L(A)$. Furthermore, $\mathrm{cr}(B) = \mathrm{cr}(A)$ because q_B has no incoming transitions and it thus forms a separate strongly connected component in B whose cycle rank is 0. From the definitions it then follows that $\text{cr}(B) = \text{cr}(A)$.

From B, we now construct the non-returning state-labeled NFA B^{sl} such that $\text{cr}(B^{\text{sl}}) \leq \text{cr}(B) \cdot |\Sigma| = \text{cr}(A) \cdot |\Sigma|$. Let $B^{\text{sl}} = (Q^{\text{sl}}, q_0^{\text{sl}}, \delta^{\text{sl}}, F^{\text{sl}})$ be defined as

- $Q^{\text{sl}} = \{q^a \mid q \in Q^B, a \in \Sigma\}$;
- $q_0^{\text{sl}} = q_B^a$, for some $a \in \Sigma$;
- $\delta^{\text{sl}} = \{(q^a, b, p^b) \mid q, p \in Q^B, a, b \in \Sigma, (q, b, p) \in \delta^B\}$; and
- $F^{\text{sl}} = \{q^a \mid q \in F^B, a \in \Sigma\}$

That is, B^{sl} contains $|\Sigma|$ copies of every state q of B, each of which captures all incoming transitions of q for one alphabet symbol. Obviously, B^{sl} is a non-returning state-labeled NFA with $L(B) = L(B^{\text{sl}})$.

We conclude by showing that $\text{cr}(B) \cdot |\Sigma| \geq \text{cr}(B^{\text{sl}})$. In the following, we abuse notation and, for a set of states P, write $B[P]$ for the subautomaton of B induced by P, defined in the obvious way. Now, for a set of states P of B, let $P^{\text{sl}} = \{q^a \mid q \in P, a \in \Sigma\}$, and observe that $(B[Q \setminus P])^{\text{sl}} = B^{\text{sl}}[Q^{\text{sl}} \setminus P^{\text{sl}}]$ always holds. We now show $\text{cr}(B) \cdot |\Sigma| \geq \text{cr}(B^{\text{sl}})$ by induction on the number of states of B. If $|Q^B| = 1$ then either the single state does not contain a loop, such that $\text{cr}(B) = \text{cr}(B^{\text{sl}}) = 0$, or the single state contains a self loop, in which case $\text{cr}(B) = 1$, and as $|Q^{\text{sl}}| = |\Sigma|$, $\text{cr}(B^{\text{sl}}) \leq |\Sigma|$ holds, and hence $\text{cr}(B) \cdot |\Sigma| \geq \text{cr}(B^{\text{sl}})$.

For the induction step, if B is acyclic, then, again, $\text{cr}(B) = \text{cr}(B^{\text{sl}}) = 0$. Otherwise, if B is strongly connected, then $\text{cr}(B) = \text{cr}(B[Q \setminus \{q\}]) + 1$, for some $q \in Q^B$. Then, $\text{cr}(B) \cdot |\Sigma| = \text{cr}(B[Q \setminus \{q\}]) \cdot |\Sigma| + |\Sigma|$, which by the induction hypothesis gives us $\text{cr}(B) \cdot |\Sigma| \geq \text{cr}(B^{\text{sl}}[Q^{\text{sl}} \setminus \{q\}^{\text{sl}}]) + |\Sigma|$. Finally, it is shown in [48] that removing a set of states P from a graph can never decrease its cycle rank by more than $|P|$. Therefore, as $|\{q\}^{\text{sl}}| = |\Sigma|$, $\text{cr}(B^{\text{sl}}[Q \setminus \{q\}^{\text{sl}}]) + |\Sigma| \geq \text{cr}(B^{\text{sl}})$, and hence $\text{cr}(B) \cdot |\Sigma| \geq \text{cr}(B^{\text{sl}})$.

Otherwise, if B consists of strongly connected components $Q_1, \ldots, Q_k$, with $k \geq 2$, then $\text{cr}(B) = \max_{i \leq k} \text{cr}(B[Q_i])$. But now, by construction, every strongly connected component V of B^{sl} is contained in Q_i^{sl}, for some $i \in [1, k]$. Then, $B^{\text{sl}}[V]$ is a minor of $B^{\text{sl}}[Q_i^{\text{sl}}]$, and hence it follows from Lemma 23 that $\text{cr}(B^{\text{sl}}) = \max_{V \in \text{SCC}(B^{\text{sl}})} \text{cr}(B^{\text{sl}}[V]) \leq \max_{i \leq k} \text{cr}(B^{\text{sl}}[Q_i^{\text{sl}}])$. Now, by induction, $\text{cr}(B[Q_i]) \cdot |\Sigma| \geq \text{cr}(B^{\text{sl}}[Q_i^{\text{sl}}])$ for all $i \in [1, k]$, from which it follows that $\text{cr}(B) \cdot |\Sigma| \geq \text{cr}(B^{\text{sl}})$. □

4.2 Succinctness w.r.t. NFAs

In this section, we study the complexity of translating extended regular expressions into NFAs. We show that such a translation can be done in exponential time, by constructing the NFA by induction on the structure of the expression.

Proposition 25. Let r be a RE($\&, \cap, \#$). An NFA A with at most $2^{|r|}$ states, such that $L(r) = L(A)$, can be constructed in time $2^{\mathcal{O}(|r|)}$.

Proof. We construct A by induction on the structure of the formula. For the base cases, $r = \varepsilon$, $r = \emptyset$, and $r = a$, for $a \in \Sigma$, and the induction cases $r = r_1 r_2$, $r = r_1 + r_2$, and r_1^* this can easily be done using standard constructions. We give the full construction for the three special operators:

- If $r = r_1 \cap r_2$, for $i \in [1, 2]$, let $A_i = (Q^i, q_0^i, \delta^i, F^i)$ accept $L(r_i)$. Then, $A = (Q, q_0, \delta, F)$ is defined as $Q = Q_1 \times Q_2$, $q_0 = (q_0^1, q_0^2)$, $F = F_1 \times F_2$, and $\delta = \{((q_1, q_2), a, (p_1, p_2)) \mid (q_1, a, p_1) \in \delta_1 \wedge (q_2, a, p_2) \in \delta_2\}$.
- If $r = r_1 \,\&\, r_2$, then A is defined exactly as for $r_1 \cap r_2$, except for δ which now equals $\{((q_1, q_2), a, (p_1, q_2)) \mid (q_1, a, p_1) \in \delta_1\} \cup \{((q_1, q_2), a, (q_1, p_2)) \mid (q_2, a, p_2) \in \delta_2\}$.
- If $r = r_1^{k,\ell}$, let A_1 accept $L(r_1)$. Then, let B_1 to B_ℓ be ℓ identical copies of A_1 with disjoint sets of states. For $i \in [1, \ell]$, let $B_i = (Q^i, q_0^i, \delta^i, F^i)$. Now, define $A = (Q, q_0^0, \delta, F)$ accepting $r_1^{k,\ell}$ as follows: $Q = \bigcup_{i \leq \ell} Q^i$, $F = \bigcup_{k \leq i \leq \ell} F^i$, and $\delta = \bigcup_{i \leq \ell} \delta^i \cup \{(q_i, a, q_0^{i+1}) \mid q_i \in Q^i \wedge \exists p_i \in F^i$ such that $(q_i, a, p_i) \in \delta_i\}$.

We note that the construction for the interleaving operator is introduced by Mayer and Stockmeyer [79], who already used it for a translation from RE($\&$) to NFAs. We argue that A contains at most $2^{|r|}$ states. For $r = r_1 \cap r_2$ or $r = r_1 \,\&\, r_2$, by induction A_1 and A_2 contain at most $2^{|r_1|}$ and $2^{|r_2|}$ states and, hence, A contains at most $2^{|r_1|} \cdot 2^{|r_2|} = 2^{|r_1|+|r_2|} \leq 2^{|r|}$ states. For $r = r_1^{k,\ell}$, similarly, A contains at most $\ell \cdot 2^{|r_1|} = 2^{|r_1|+\log \ell} \leq 2^{|r|}$ states. Furthermore, as the intermediate automata never have more than $2^{|r|}$ states and we have to do at most $|r|$ such constructions, the total construction can be done in time $2^{\mathcal{O}(|r|)}$. □

This exponential size increase can not be avoided for any of the classes. For RE($\#$) this is witnessed by the expression $a^{2^n,2^n}$ and for RE($\&$) by the expression $a_1 \,\&\, \cdots \,\&\, a_n$, as already observed by Kilpelainen and Tuhkanen [64] and Mayer and Stockmeyer [79], respectively. For RE($\cap$), a $2^{\Omega(\sqrt{n})}$ lower bound has already been reported in [90]. The present tighter statement, however, will follow from Theorem 29 and the fact that any NFA with n states can be translated into a DFA with 2^n states (Theorem 2(2)).

Proposition 26. For any $n \in \mathbb{N}$, there exist an RE(#) $r^{\#}$, an RE($\cap$) $r^{\cap}$, and an RE(&) $r^{\&}$, each of size $\mathcal{O}(n)$, such that any NFA accepting $r^{\#}$, $r^{\cap}$, or $r^{\&}$ contains at least 2^n states.

4.3 Succinctness w.r.t. DFAs

In this section, we study the complexity of translating extended regular expressions into DFAs. First, from Proposition 25 and the fact that any NFA with n states can be translated into a DFA with 2^n states in exponential time (Theorem 2(2)), we can immediately conclude the following.

Proposition 27. Let r be a RE($\&, \cap, \#$). A DFA A with at most $2^{2^{|r|}}$ states, such that $L(r) = L(A)$, can be constructed in time $2^{2^{\mathcal{O}(|r|)}}$.

We show that, for each of the classes RE(#), RE($\cap$), or RE(&), this double exponential size increase can not be avoided.

Proposition 28. For any $n \in \mathbb{N}$ there exists an RE(#) r_n of size $\mathcal{O}(n)$ such that any DFA accepting $L(r_n)$ contains at least 2^{2^n} states.

Proof. Let $n \in \mathbb{N}$ and define $r_n = (a+b)^*a(a+b)^{2^n,2^n}$. Here, r_n is of size $\mathcal{O}(n)$ since the integers in the numerical predicate are stored in binary. We show that any DFA $A = (Q, q_0, \delta, F)$ accepting $L(r_n)$ has at least 2^{2^n} states. Towards a contradiction, suppose that A has less than 2^{2^n} states and consider all strings of length 2^n containing only a's and b's. As there are exactly 2^{2^n} such strings, and A contains less than 2^{2^n} states, there must be two different such strings w, w' and a state q of A such that both $(q_0, w, q) \in \delta^*$ and $(q_0, w', q) \in \delta^*$. But now, as $w \neq w'$, there exists some $i \in [1, 2^n]$ such that the ith position of w contains an a, and the ith position of w' contains a b (or the other way around, but that is identical). Therefore, $wa^i \in L(r_n)$, and $w'a^i \notin L(r_n)$ but wa^i and $w'a^i$ are either both accepted or both not accepted by A, and hence $L(r_n) \neq L(A)$, which gives us the desired contradiction. $\square$

We now move to regular expressions extended with the intersection operator. The succinctness of RE($\cap$) with respect to DFAs can be obtained along the same lines as the simulation of exponential space turing machines by RE($\cap$) by Fürer [35].

Theorem 29. For any $n \in \mathbb{N}$ there exists an RE($\cap$) $r_n^{\cap}$ of size $\mathcal{O}(n)$ such that any DFA accepting $L(r_n^{\cap})$ contains at least 2^{2^n} states.

Proof. Let $n \in \mathbb{N}$. We start by describing the language $\mathcal{G}_n$ which will be used to establish the lower bound. This will be a variation of the following language over the alphabet $\{a, b\}$: $\{ww \mid |w| = 2^n\}$. It is well-known that this language is hard to describe by a DFA. However, to define it succinctly by an RE($\cap$) expression, we need to add some additional information to it.

Thereto, we first define a *marked number* as a string over the alphabet $\{0, 1, \overline{0}, \overline{1}\}$ defined by the regular expression $(0 + 1)^*\overline{1}\overline{0}^* + \overline{0}^*$, i.e., a binary number in which the rightmost 1 and all following 0's are marked. Then, for any $i \in [0, 2^n - 1]$ let enc(i) denote the n-bit marked number encoding i. These marked numbers were introduced in [35], where the following is observed: if $i, j \in [0, 2^n - 1]$ are such that $j = i + 1 \pmod{2^n}$, then the bits of i and j which are different are exactly the marked bits of j. For instance, for $n = 2$, enc(1) $= 0\overline{1}$ and enc(2) $= \overline{1}\overline{0}$ and they differ in both bits as both bits of enc(2) are marked. Further, let $\text{enc}^R(i)$ denote the reversal of enc(i).

Now, for a string $w = a_0a_1 \dots a_{2^n-1}$, define

$$\text{enc}(w) = \text{enc}^R(0)a_0\text{enc}(0)\$\text{enc}^R(1)a_1\text{enc}(1)\$ \cdots \text{enc}^R(2^n - 1)a_{2^n-1}\text{enc}(2^n - 1)$$

and, finally, define

$$\mathcal{G}_n = \{\#\text{enc}(w)\#\text{enc}(w) \mid w \in L((a + b)^*) \wedge |w| = 2^n\}.$$

For instance, for $n = 2$, and $w = abba$, $\text{enc}(w) = \overline{00}a\overline{00}\$\overline{1}0b0\overline{1}\$\overline{01}b\overline{10}\$\overline{1}1a1\overline{1}$ and hence $\#\overline{00}a\overline{00}\$\overline{1}0b0\overline{1}\$\overline{01}b\overline{10}\$\overline{1}1a1\overline{1}\#\overline{00}a\overline{00}\$\overline{1}0b0\overline{1}\$\overline{01}b\overline{10}\$\overline{1}1a1\overline{1} \in \mathcal{G}_2$.

Now, lets consider $\overline{\mathcal{G}_n}$, the complement of $\mathcal{G}_n$. Using a standard argument, similar to the in one in the proof of Proposition 28, it is straightforward to show that any DFA accepting $\overline{\mathcal{G}_n}$ must contain at least 2^{2^n} states. We conclude by showing that we can construct a regular expression $r_n^{\cap}$ of size $\mathcal{O}(n)$ defining $\overline{\mathcal{G}_n}$.

Note that $\Sigma = \{0, \overline{0}, 1, \overline{1}, a, b, \$, \#\}$. We define $N = \{0, \overline{0}, 1, \overline{1}\}$, $S = \{a, b\}$, $D = \{\$, \#\}$ and for any set U, and $\sigma \in U$, let $U_\sigma = U \setminus \{\sigma\}$. Now, we construct a set of expressions, each capturing a possible mistake in a string. Then, $r_n^{\cap}$ is simply the disjunction of these expressions. The expressions are as follows. All strings which do not start with #:

$$\varepsilon + \Sigma_\# \Sigma^* \,.$$

All strings in which two symbols at a distance $n + 1$ do not match:

$$\Sigma^*(N\Sigma^n(S + D) + S\Sigma^n(S + N) + D\Sigma^n(D + N))\Sigma^* \,.$$

All strings which do not end with $\text{enc}(2^n - 1) = 1^{n-1}\overline{1}$:

$$\Sigma^*(\Sigma_{\overline{1}} + \Sigma_1\Sigma^{1,n-1}) \,.$$

All strings in which # occurs before any number other than $\mathrm{enc}^R(0)$ or where \$ occurs before $\mathrm{enc}^R(0)$:

$$\Sigma^*(\$\overline{0}^n + \#(\Sigma^{0,n-1}\Sigma_{\overline{0}}))\Sigma^* .$$

All strings which contain more or less than 2 #-symbols

$$\Sigma_{\#}^*(\# + \varepsilon)\Sigma_{\#}^* + \Sigma^*\#\Sigma^*\#\Sigma^*\#\Sigma^* .$$

All strings which contain a (non-reversed) binary number which is not correctly marked:

$$\Sigma^* SN^*((0+1)(\overline{0} + \$ + \#) + (\overline{0} + \overline{1})N_{\overline{0}})\Sigma^* .$$

All strings in which the binary encodings of two numbers surrounding an a or b are not each others reverse. Thereto, we first define expressions r_i, for all $i \in [0,n]$, such that $L(r_i) = \{w\sigma w' \mid \sigma \in S \wedge w, w' \in \Sigma^* \wedge |w| = |w'| \wedge |w| \leq i\}$, inductively as follows: $r_0 = S$ and, for all $j \in [1,n]$, $r_j = r_0 + \Sigma r_{j-1}\Sigma$. Then, the following is the desired expression:

$$\Sigma^*(r_n \cap \bigcup_{\sigma \in N} \sigma\Sigma^*\Sigma_\sigma)\Sigma^* .$$

All strings in which the binary encodings of two numbers surrounding a \$ or # do not differ by exactly one, i.e., there is a substring of the form $\mathrm{enc}(i)\$\mathrm{enc}^R(j)$ or $\mathrm{enc}(i)\#\mathrm{enc}^R(j)$ such that $j \neq i + 1 \pmod{2^n}$. Exactly as above, we can inductively define r'_n such that $L(r'_n) = \{w\sigma w' \mid \sigma \in D \wedge w, w' \in \Sigma^* \wedge |w| = |w'| \wedge |w| \leq n\}$. Then, we obtain:

$$\Sigma^*(r'_n \cap ((0 + \overline{0})\Sigma^*(\overline{0} + 1) + (1 + \overline{1})\Sigma^*(\overline{1} + 0)))\Sigma^* .$$

All strings in which two a or b symbols which should be equal are not equal. We now define expressions s_i, for all $i \in [0,n]$ such that $L(s_i) = \{wu\#vw^R \mid u, v, w \in \Sigma^* \wedge |w| = i\}$. By induction, $s_0 = \Sigma^*\#\Sigma^*$, and for all $i \in [1,n]$, $s_i = \Sigma s_{i-1}\Sigma \cap \bigcup_{\sigma \in \Sigma} \sigma\Sigma^*\sigma$. Then, the following is the desired expression:

$$\Sigma^*(as_nb + bs_na)\Sigma^* .$$

Now, a string is not in $\mathcal{G}_n$ if and only if it is accepted by at least one of the previous expressions. Hence, $r_n^{\cap}$, defined as the disjunction of all these expressions, defines exactly $\overline{\mathcal{G}_n}$. Furthermore, notice that all expressions, including the inductively defined ones, are of size $\mathcal{O}(n)$, and hence $r_n^{\cap}$ is also of size $\mathcal{O}(n)$. This concludes the proof. □

We can now extend the results for RE($\cap$) to RE(&). We do this by using a technique of Mayer and Stockmeyer [79] which allows, in some sense, to simulate an RE($\cap$) expression by an RE(&) expression. We illustrate their technique in a simple case, for an expression $r = r_1 \cap r_2$, where r_1 and r_2 are standard regular expressions, not containing intersection operators. Let c be a symbol not occurring in r, and for $i = 1, 2$, define r_i^c as the expression obtained from r_i by replacing any symbol a in r_i by ac. Then, it is easily seen that a string $a_0 \cdots a_n \in L(r_i)$ if and only if $a_0 c \cdots a_n c \in L(r_i)$. Now, consider the expression $r^c = r_1^c \,\&\, r_2^c$. Then, a string $a_0 \cdots a_n \in L(r) = L(r_1) \cap L(r_2)$ if and only if $a_0 c \cdots a_n c \in L(r_1^c) \cap L(r_2^c)$ if and only if $a_0^2 c^2 \cdots a_n^2 c^2 \in L(r^c)$. So, not all strings defined by r^c are of the form $a_0^2 c^2 \cdots a_n^2 c^2$, but the set of strings of the form $a_0^2 c^2 \cdots a_n^2 c^2$ defined by r^c corresponds exactly to the set of strings defined by r. In this sense, r^c simulates r.

The latter technique can be extended to general RE($\cap$) and RE(&). To formally define this, we need some notation. Let $w = a_0 \cdots a_n$ be a string over an alphabet Σ, and let c be a symbol not in Σ. Then, for any $i \in \mathbb{N}$, define $\text{pump}_i(w) = a_0^i c^i a_1^i c^i \cdots a_k^i c^i$.

Lemma 30 ([79])**.** Let r be an RE($\cap$) containing k $\cap$-operators. Then, there exists an RE(&) s of size at most $|r|^2$ such that for any $w \in \Sigma^*$, $w \in L(r)$ if and only if $\text{pump}_k(w) \in L(s)$.

Using this lemma, we can now prove the following theorem.

Theorem 31. For any $n \in \mathbb{N}$ there exists an RE(&) $r_n^{\&}$ of size $\mathcal{O}(n^2)$ such that any DFA accepting $L(r_n^{\&})$ contains at least 2^{2^n} states.

Proof. Let $n \in \mathbb{N}$ and consider the expression $r_n^{\cap}$ of size $\mathcal{O}(n)$ constructed in Theorem 29 such that any DFA accepting $L(r_n^{\cap})$ contains at least 2^{2^n} states. Now, let $r_n^{\&}$ be the regular expression *simulating* $r_n^{\cap}$ obtained from Lemma 30, such that for some $k \in \mathbb{N}$ and any $w \in \Sigma^*$, $w \in L(r_n^{\cap})$ if and only if $\text{pump}_k(w) \in L(r_n^{\&})$. Then, $r_n^{\&}$ is of size $\mathcal{O}(n^2)$ and, exactly as before, it is straightforward to show that any DFA accepting $L(r_n^{\&})$ contains at least 2^{2^n} states. $\square$

4.4 Succinctness w.r.t. Regular Expressions

In this section, we study the translation of extended regular expressions to (standard) regular expressions. First, for the class RE(#) it has already been shown by Kilpelainen and Tuhkanen [64] that this translation can be done in exponential time, and that an exponential size increase can not be avoided. Furthermore, from Proposition 25 and the fact that any NFA with n states can be translated into a regular expression in time $2^{\mathcal{O}(n)}$ (Theorem 2(1)) it immediately follows that:

Proposition 32. Let r be a RE(&, $\cap$, #). A regular expression s equivalent to r can be constructed in time $2^{2^{\mathcal{O}(|r|)}}$

Furthermore, from Theorem 20 it follows that in a translation from RE($\cap$) to standard regular expressions, a double exponential size increase can not be avoided.

Hence, it only remains to show a double exponential lower bound on the translation from RE(&) to standard regular expressions, which is exactly what we will do in the rest of this section. Thereto, we proceed in several steps and define several families of languages. First, we revisit the family of languages $(\mathcal{K}_n)_{n\in\mathbb{N}}$, on which all following languages will be based, and establish its star height. The star height of languages will be our tool for proving lower bounds on the size of regular expressions defining these languages. Then, we define the family $(\mathcal{L}_n)_{n\in\mathbb{N}}$ which is a binary encoding of $(\mathcal{K}_n)_{n\in\mathbb{N}}$, different from the binary encoding introduced in Section 3.1, and show that these languages can be defined as the intersection of small regular expressions.

Finally, we define the family $(\mathcal{M}_n)_{n\in\mathbb{N}}$ which is obtained by simulating the intersection of the previously obtained regular expressions by the interleaving of related expressions, similar to the simulation of RE($\cap$) by RE(&) in Section 4.3. Bringing everything together, this then leads to the desired result: a double exponential lower bound on the translation of RE(&) to RE.

4.4.1 $\mathcal{K}_n$: The Basic Language

Recall from Section 3.1 that the language $\mathcal{K}_n$, for any $n \in \mathbb{N}$, consists of all strings of the form $a_{0,i_1}a_{i_1,i_2}\cdots a_{i_k,n-1}$ where $k \in \mathbb{N}\cup\{0\}$. We now determine the star height of $\mathcal{K}_n$.

Lemma 33. For any $n \in \mathbb{N}$, $\mathrm{sh}(\mathcal{K}_n) = n$.

Proof. We start by observing that, for any $n \in \mathbb{N}$, the language $\mathcal{K}_n$ is bideterministic. Indeed, the inverse of the DFA $A_n^{\mathcal{K}}$ accepting $\mathcal{K}_n$ is again deterministic as every transition is labeled with a different symbol. Furthermore, $A_n^{\mathcal{K}}$ is the minimal trimmed DFA accepting $\mathcal{K}_n$. Hence, by Theorem 24(3), $\mathrm{sh}(\mathcal{K}_n) = \mathrm{cr}(A_n^{\mathcal{K}})$. We conclude by showing that, for any $n \in \mathbb{N}$, $\mathrm{cr}(A_n^{\mathcal{K}}) = n$.

Thereto, first observe that the graph underlying $A_n^{\mathcal{K}}$ is the complete graph (including self-loops) on n nodes, which we denote by K_n. We proceed by induction on n. For $n = 1$, the graph is a single node with a self loop and hence by definition $\mathrm{cr}(K_1) = 1$. For the inductive step, suppose that $\mathrm{cr}(K_n) = n$ and consider K_{n+1} with node set V_{n+1}. Since V_{n+1} consists of only one strongly connected component, $\mathrm{cr}(K_{n+1}) = 1 + \min_{v\in V_{n+1}}\{\mathrm{cr}(K_{n+1}[V_{n+1} \setminus \{v\}])\}$. However, for any $v \in V_{n+1}$, $K_{n+1}[V_{n+1} \setminus \{v\}]$ is isomorphic to K_n, and hence by the induction hypothesis $\mathrm{cr}(K_{n+1}) = n + 1$. ☐

4.4.2 $\mathcal{L}_n$: Binary Encoding of $\mathcal{K}_n$

In this section we want to construct a set of small regular expressions such that any expression defining their intersection must be large (that is, of double exponential size). Ideally, we would like to use the languages of the family $(\mathcal{K}_n)_{n\in\mathbb{N}}$ for this as we have shown in the previous section that they have a large star height, and thus by Theorem 22 can not be defined by small expressions. Unfortunately, this is not possible as the alphabet of $(\mathcal{K}_n)_{n\in\mathbb{N}}$ grows quadratically with n.

Therefore, we will introduce in this section the family of languages $(\mathcal{L}_n)_{n\in\mathbb{N}}$ which is a binary encoding of $(\mathcal{K}_n)_{n\in\mathbb{N}}$ over a fixed alphabet. Thereto, let $n \in \mathbb{N}$ and recall that $\mathcal{K}_n$ is defined over the alphabet $\Sigma_n = \{a_{i,j} \mid i, j \in [0, n-1]\}$. Now, for $a_{i,j} \in \Sigma_n$, define the function ρ_n as

$$\rho_n(a_{i,j}) = \#\mathrm{enc}(j)\$\mathrm{enc}(i)\triangle\mathrm{enc}(i+1)\triangle \cdots \triangle\mathrm{enc}(n-1)\triangle,$$

where $\mathrm{enc}(k)$, for $k \in \mathbb{N}$, denotes the $\lceil\log(n)\rceil$-bit marked number encoding k as defined in the proof of Theorem 29. So, the encoding starts by the encoding of the second index, followed by an ascending sequence of encodings of all numbers from the first index to $n-1$. We extend the definition of ρ_n to strings in the usual way: $\rho_n(a_{0,i_1} \cdots a_{i_{k-1},n-1}) = \rho_n(a_{0,i_1}) \cdots \rho_n(a_{i_k,n-1})$. We are now ready to define $\mathcal{L}_n$.

Definition 34. Let $\Sigma = \{0, 1, \overline{0}, \overline{1}, \$, \#, \triangle\}$. For $n \in \mathbb{N}$, $\mathcal{L}_n = \{\rho_n(w) \mid w \in \mathcal{K}_n\}$.

For instance, for $n = 3$, $a_{0,1}a_{1,2} \in \mathcal{K}_3$ and hence $\rho_3(a_{0,1}a_{1,2}) = \#0\overline{1}\$\overline{0}\overline{0}\triangle 0\overline{1}$ $\triangle\overline{1}\overline{0}\triangle\#\overline{1}\overline{0}\$0\overline{1}\triangle\overline{1}\overline{0}\triangle \in \mathcal{L}_3$. We now show that this encoding does not affect the star height.

Lemma 35. For any $n \in \mathbb{N}$, $\mathrm{sh}(\mathcal{L}_n) = n$.

Proof. Let $n \in \mathbb{N}$. We first show that $\mathrm{sh}(\mathcal{L}_n) \leq n$. By Lemma 33, $\mathrm{sh}(\mathcal{K}_n) = n$, and hence there exists a regular expression $r_{\mathcal{K}}$, with $L(r_{\mathcal{K}}) = \mathcal{K}_n$ and $\mathrm{sh}(r_{\mathcal{K}}) = n$. Let $r_{\mathcal{L}}$ be the regular expression obtained from $r_{\mathcal{K}}$ by replacing every symbol $a_{i,j}$ by $\rho_n(a_{i,j})$. Obviously, $L(r_{\mathcal{L}}) = \mathcal{L}_n$ and $\mathrm{sh}(r_{\mathcal{L}}) = n$, and hence $\mathrm{sh}(\mathcal{L}_n) \leq n$.

We now show that $\mathrm{sh}(\mathcal{L}_n) \geq n$. The proof is along the same lines as the proof of Lemma 33. We first show that $\mathcal{L}_n$ is bideterministic and can then determine its star height by looking at the minimal DFA accepting it. In fact, the reason we use a slightly involved encoding of $\mathcal{K}_n$, different from the one in Chapter 3, is precisely the bideterminism property.

To show that $\mathcal{L}_n$ is bideterministic, we now construct the minimal DFA $A_n^{\mathcal{L}}$ accepting $\mathcal{L}_n$. Here, $A_n^{\mathcal{L}}$ will consist of n identical subautomata B_n^0 to B_n^{n-1}

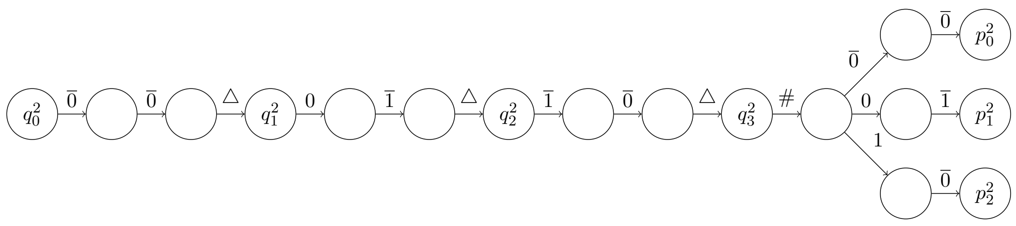


Figure 4.2: The automaton B_3^2.

defined as follows. For any $i \in [0, n-1]$, $B_n^i = (Q^i, q_n^i, \delta_i, F^i)$ is the smallest automaton for which Q^i contains distinct states $q_0^i, \ldots, q_n^i$ and $p_0^i, \ldots, p_{n-1}^i$ such that for any $j \in [0, n-1]$,

- $(q_j^i, \mathrm{enc}(j)\triangle, q_{j+1}^i) \in \delta_i^*$, and for all $w \neq \mathrm{enc}(j)\triangle$, $(q_j^i, w, q_{j+1}^i) \notin \delta_i^*$; and
- $(q_n^i, \#\mathrm{enc}(j), p_j^i) \in \delta_i^*$, and for all $w \neq \#\mathrm{enc}(j)$, $(q_n^i, w, p_j^i) \notin \delta_i^*$.

As an example, Figure 4.2 shows B_3^2. Now, $A_n^{\mathcal{L}} = (Q, q_n^0, \delta, F)$ is defined as $Q = \bigcup_{i<n} Q^i$, $F = \{q_n^{n-1}\}$ and $\delta = \bigcup_{i<n} \delta_i \cup \{(p_j^i, \$, q_i^j) \mid i, j \in [0, n-1]\}$. Figure 4.3 shows $A_3^{\mathcal{L}}$.

To see that $A_n^{\mathcal{L}}$ indeed accepts $\mathcal{L}_n$, notice that after reading a substring $\#\mathrm{enc}(i)$, for some i, the automaton moves to sub-automaton B_n^i. Then, after passing through B_n^i which ends by reading a new substring $\#\mathrm{enc}(j)$, the automaton moves to state q_i^j of sub-automaton B_n^j. This ensures that the subsequent ascending sequence of numbers starts with $\mathrm{enc}(i)$. Hence, the automaton checks correctly whether the numbers which should be equal, are equal.

Furthermore, $A_n^{\mathcal{L}}$ is bideterministic and minimal. To see that it is bideterministic, notice that all states except the q_j^i only have one incoming transition. Furthermore, the q_j^i each have exactly two incoming transitions, one labeled with \$ and one labeled with $\triangle$. Minimality follows immediately from the fact that $A_n^{\mathcal{L}}$ is both trimmed and bideterministic.

Now, since $\mathcal{L}_n$ is bideterministic and $A_n^{\mathcal{L}}$ is the minimal trimmed DFA accepting $\mathcal{L}_n$, it follows from Theorem 24(3) that $\mathrm{sh}(\mathcal{L}_n) = \mathrm{cr}(A_n^{\mathcal{L}})$. Therefore, it suffices to show that $\mathrm{cr}(A_n^{\mathcal{L}}) \geq n$. Thereto, observe that $A_n^{\mathcal{K}}$, the minimal DFA accepting $\mathcal{K}_n$, is a minor of $A_n^{\mathcal{L}}$. Indeed, we can easily contract edges and remove nodes from $A_n^{\mathcal{K}}$ such that the only remaining nodes are q_n^0 to q_n^{n-1}, and such that they form a complete graph. Now, since it is shown in Lemma 33 that $\mathrm{cr}(A_n^{\mathcal{K}}) = n$ and by Lemma 23 and the fact that $A_n^{\mathcal{K}}$ is a minor of $A_n^{\mathcal{L}}$,

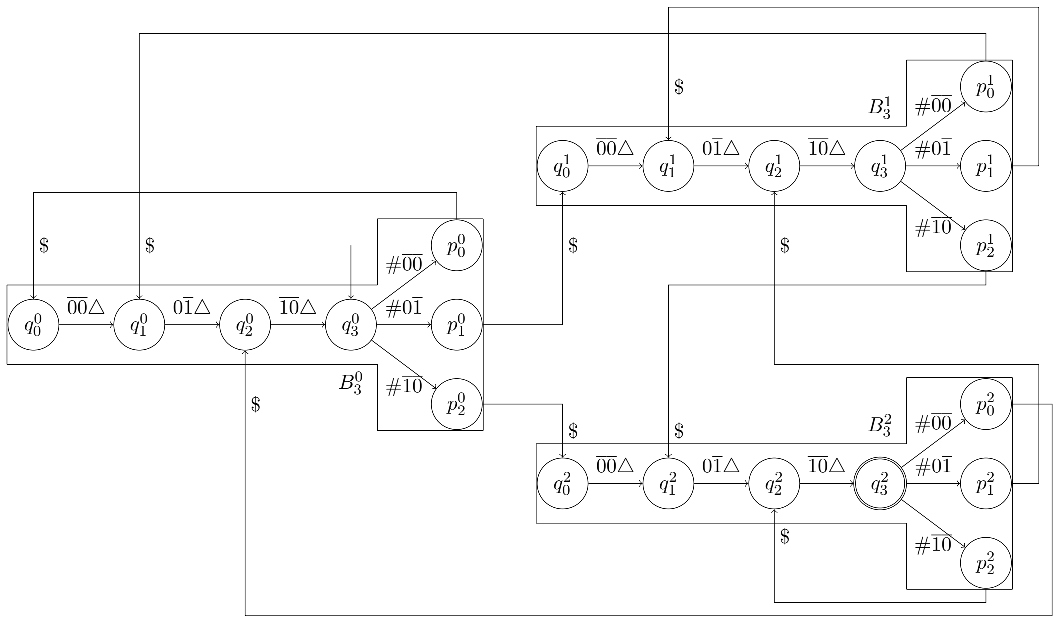


Figure 4.3: The DFA $A_3^{\mathcal{L}}$, accepting $\mathcal{L}_3$.

we know that $\mathrm{cr}(A_n^{\mathcal{L}}) \geq \mathrm{cr}(A_n^{\mathcal{K}})$, it follows that $\mathrm{sh}(\mathcal{L}_n) = \mathrm{cr}(A_n^{\mathcal{L}}) \geq n$. This completes the proof. □

Furthermore, it can be shown that $\mathcal{L}_n$ can be described as the intersection of a set of small regular expressions.

Lemma 36. For every $n \in \mathbb{N}$, there are regular expressions $r_1, \ldots, r_m$, with $m = 4n + 3$, each of size $\mathcal{O}(n)$, such that $\bigcap_{i \leq m} L(r_i) = \mathcal{L}_{2^n}$.

Proof. Let $n \in \mathbb{N}$, and recall that $\mathcal{L}_{2^n}$ is defined over the alphabet $\Sigma = \{0, 1, \overline{0}, \overline{1}, \$, \#, \triangle\}$. Let $N = \{0, 1, \overline{0}, \overline{1}\}$, $D = \{\$, \#, \triangle\}$ and for any $\sigma \in \Sigma$, let $\Sigma_\sigma = \Sigma \setminus \{\sigma\}$. The expressions are as follows.
The format of the string has to be correct:

$$(\# N^n \$ N^n \triangle (N^n \triangle)^+)^+ .$$

Every number should be properly marked:

$$(D^+((0+1)^* \overline{1} \overline{0}^* + \overline{0}^*))^* .$$

Every number before a \$ should be equal to the number following the next \$. We define two sets of regular expressions. First, for all $i \in [0, n-1]$, the

$(i+1)$th bit of the number before an even \$ should be equal to the $(i+1)$th bit of the number after the next \$.

$$(\#N^i \bigcup_{\sigma \in N} (\sigma \Sigma_\$^* \$ \Sigma_\$^* \$ N^i \sigma) \Sigma_\#^*)^* (\varepsilon + \#\Sigma_\#^*)\,.$$

Second, for all $i \in [0, n-1]$, the $(i+1)$th bit of the number before an odd \$ should be equal to the $(i+1)$th bit of the number after the next \$.

$$\#\Sigma_\#^* (\#N^i \bigcup_{\sigma \in N} (\sigma \Sigma_\$^* \$ \Sigma_\$^* \$ N^i \sigma) \Sigma_\#^*)^* (\varepsilon + \#\Sigma_\#^*)\,.$$

Every two (marked) numbers surrounding a $\triangle$ should differ by exactly one. Again, we define two sets of regular expressions. First, for all $i \in [0, n-1]$, the $(i+1)$th bit of the number before an even $\triangle$ should properly match the $(i+1)$th bit of the next number:

$$(\#\Sigma_\$^*((\triangle+\$)\Sigma^i((0+\overline{0})\Sigma_\#^n(\overline{1}+0)+(1+\overline{1})\Sigma_\#^n(\overline{0}+1))\Sigma^{n-i-1})^*((\triangle+\$)\Sigma^n+\varepsilon)\triangle)^*.$$

Second, for all $i \in [0, n-1]$, the $(i+1)$th bit of the number before an odd $\triangle$ should properly match the $(i+1)$th bit of the next number:

$$(\#\Sigma_\$^*\$\Sigma^n(\triangle\Sigma^i((0+\overline{0})\Sigma_\#^n(\overline{1}+0)+(1+\overline{1})\Sigma_\#^n(\overline{0}+1))\Sigma^{n-i-1})^*(\triangle\Sigma^n+\varepsilon)\triangle)^*.$$

Every ascending sequence of numbers should end with $\text{enc}(2^n-1) = 1^{n-1}\overline{1}$:

$$(\#\Sigma_\#^* 1^{n-1}\overline{1}\triangle)^*\,.$$

□

4.4.3 $\mathcal{M}_n$: Succinctness of RE(&)

In this section, we will finally show that RE(&) are double exponentially more succinct than standard regular expressions. We do this by simulating the intersection of the regular expressions obtained in the previous section, by the interleaving of related expressions, similar to the simulation of RE($\cap$) by RE(&) in Section 4.3. This approach will partly yield the following family of languages. For any $n \in \mathbb{N}$, define

$$\mathcal{M}_n = \{\text{pump}_{4\lceil \log n \rceil + 3}(w) \mid w \in \mathcal{L}_n\}\,.$$

As $\mathcal{M}_n$ is very similar to $\mathcal{L}_n$, we can easily extend the result on the star height of $\mathcal{L}_n$ (Lemma 35) to $\mathcal{M}_n$:

Lemma 37. For any $n \in \mathbb{N}$, $\text{sh}(\mathcal{M}_n) = n$.

Proof. The proof is along the same lines as the proof of Lemma 35. Again, $\mathcal{M}_n$ is bideterministic as witnessed by the minimal DFA $A_n^{\mathcal{M}}$ accepting $\mathcal{M}_n$. In fact, $A_n^{\mathcal{M}}$ is simply obtained from $A_n^{\mathcal{L}}$ by replacing each transition over a symbol a by a sequence of states reading $a^m c^m$, with $m = 4\lceil \log n \rceil + 3$. Some care has to be taken, however, for the states q_j^i, as these have two incoming transitions: one labeled with \$ and one labeled with $\triangle$. Naively creating these two sequences of states leading to q_j^i, we obtain a non-minimal DFA. However, if we merge the last m states of these two sequences (the parts reading c^m), we obtain the minimal trimmed DFA $A_n^{\mathcal{M}}$ accepting $\mathcal{M}_n$. Then, the proof proceeds exactly as the proof of Lemma 35. □

However, the language we will eventually define will not be exactly $\mathcal{M}_n$. Therefore, we need an additional lemma, for which we first introduce some notation. For $k \in \mathbb{N}$, and an alphabet Σ, we define $\Sigma^{(k)}$ to be the language defined by the expression $(\bigcup_{\sigma \in \Sigma} \sigma^k)^*$, i.e., all strings which consist of a sequence of blocks of identical symbols of length k. Further, for a language L, define $\text{index}(L) = \max\{i \mid i \in \mathbb{N} \wedge \exists w, w' \in \Sigma^*, a \in \Sigma \text{ such that } wa^iw' \in L\}$. Notice that index$(L)$ can be infinite. However, we will only be interested in languages for which it is finite, as in the following lemma.

Lemma 38. Let L be a regular language, and $k \in \mathbb{N}$, such that $\text{index}(L) \leq k$. Then, $\text{sh}(L) \cdot |\Sigma| \geq \text{sh}(L \cap \Sigma^{(k)})$.

Proof. Let L be a regular language, and $k \in \mathbb{N}$, such that $\text{index}(L) \leq k$. We show that, given a non-returning state-labeled NFA A accepting L, we can can construct an NFA B such that $L(B) = L \cap \Sigma^{(k)}$ and B is a minor of A.

We show how this implies the lemma. Let A be any non-returning state-labeled NFA of minimal cycle rank accepting L, and let B be as constructed above. First, since B is a minor of A it follows from Lemma 23 that $\text{cr}(B) \leq \text{cr}(A)$. Furthermore, by Theorem 24(1), $\text{cr}(B) \geq \text{sh}(L(B)) = \text{sh}(L \cap \Sigma^{(k)})$, and hence $\text{cr}(A) \geq \text{sh}(L \cap \Sigma^{(k)})$. Now, since A is a non-returning state-labeled NFA of minimal cycle rank, we can conclude from Theorem 24(2) that $\text{sh}(L) \cdot |\Sigma| \geq \text{cr}(A)$, and thus $\text{sh}(L) \cdot |\Sigma| \geq \text{sh}(L \cap \Sigma^{(k)})$.

We conclude by giving the construction of B. Thereto, let $A = (Q, q_0, \delta, F)$ be a non-returning state-labeled NFA. Now, for any $q \in Q$, and $a \in \Sigma$, define the function $\text{in}_a(q)$ as follows:

$$\text{in}_a(q) = \max\{i \mid i \in \mathbb{N} \cup \{0\} \wedge \exists w \in \Sigma^* \text{ such that } (q_0, wa^i, q) \in \delta^*\}\,.$$

Notice that as i can equal zero and index(L) is finite, $\text{in}_a(q)$ is well defined for any state which is not useless. Intuitively, $\text{in}_a(q)$ represents the maximal number of a symbols which can be read when entering state q.

Now, the following algorithm transforms A, accepting L, into B, accepting $L \cap \Sigma^{(k)}$. Repeat the following two steps, until no more changes are made:

1. Apply one of the following rules, if possible:
   (a) If exists $q, q' \in Q$, $a \in \Sigma$, with $(q, a, q') \in \delta$, such that $\text{in}_a(q') > \text{in}_a(q) + 1 \Rightarrow$ remove (q, a, q') from δ.
   (b) If exists $q, q' \in Q$, $a \in \Sigma$, with $(q, a, q') \in \delta$ and $q \neq q_0$, such that $\text{in}_{\text{symbol}(q)}(q) < k$ and $\text{symbol}(q) \neq a \Rightarrow$ remove (q, a, q') from δ.
   (c) If exists $q \in F$, $q \neq q_0$, such that $\text{in}_{\text{symbol}(q)}(q) < k \Rightarrow$ make q non-final, i.e., remove q from F.
2. Remove all useless states from A, and recompute $\text{in}_a(q)$ for all $q \in Q$, $a \in \Sigma$.

It remains to show that B, the automaton obtained when no more rules can be applied, is the desired automaton. That is, that B is a minor of A and that $L(B) = L(A) \cap \Sigma^{(k)}$. It is immediate that B is a minor of A since B is obtained from A by only removing transitions or states.

To show that $L(B) = L(A) \cap \Sigma^{(k)}$, we first prove that $L(A) \cap \Sigma^{(k)} \subseteq L(B)$. Thereto, let $A_1, \dots, A_n$, with $A_1 = A$ and $A_n = B$, be the sequence of NFAs produced by the algorithm, where each A_i is obtained from A_{i-1} by applying exactly one rule and possibly removing useless states. It suffices to show that for all $i \in [1, n-1]$, $L(A_i) \cap \Sigma^{(k)} \subseteq L(A_{i+1}) \cap \Sigma^{(k)}$, as $L(A) \cap \Sigma^{(k)} \subseteq L(B) \cap \Sigma^{(k)} \subseteq L(B)$ then easily follows.

Before proving $L(A_i) \cap \Sigma^{(k)} \subseteq L(A_{i+1}) \cap \Sigma^{(k)}$, we introduce some notation. For any $q \in Q$, $a \in \Sigma$, we define

$$\text{out}_a(q) = \max\{i \mid i \in \mathbb{N} \cup \{0\} \wedge \exists w \in \Sigma^*, q_f \in F \text{ such that } (q, a^i w, q_f) \in \delta^*\}.$$

Here, $\text{out}_a(q)$ is similar to $\text{in}_a(q)$ and represents the maximal number of a's which can be read when leaving q. Since of course $L(A_i) \subseteq L(A)$ holds for all $i \in [1, n]$, it also holds that $\text{index}(A_i) \leq k$. Therefore, for any A_i, any state q of A_i and any $a \in \Sigma$, it holds that

$$\text{in}_a(q) + \text{out}_a(q) \leq k\,. \tag{4.1}$$

We are now ready to show that $L(A_i) \cap \Sigma^{(k)} \subseteq L(A_{i+1}) \cap \Sigma^{(k)}$. Thereto, we prove that for any $w \in L(A_i) \cap \Sigma^{(k)}$, any accepting run of A_i on w is still an accepting run of A_{i+1} on w. More precisely, we prove for every rule separately that if A_{i+1} is obtained from A_i by applying this rule, then the assumption that the removed state or transition is used in an accepting run of A_i on w leads to a contradiction.

For rule (1a), suppose transition (q, a, q') is removed but (q, a, q') occurs in an accepting run of A_i on w, i.e., $w = uav$ for $u, v \in \Sigma^*$, $(q_0, u, q) \in \delta^*$ and $(q', v, q_f) \in \delta^*$, for some $q_f \in F$. Since $w \in \Sigma^{(k)}$, there exists a $j \in [0, n-1]$ and $u', v' \in \Sigma^*$ such that $u = u'a^j$ and $v = a^{k-j-1}v'$. It immediately follows that $\text{in}_a(q) \geq j$, and $\text{out}_a(q') \geq k - j - 1$. Then, since $\text{in}_a(q') > \text{in}_a(q) + 1$, we have $\text{in}_a(q') + \text{out}_a(q') > j + k - j - 1 + 1 = k$. However, this contradicts equation (4.1).

For rules (1b) and (1c), we describe the obtained contradictions less formal. For rule (1b), if the transition (q, a, q'), with $\text{symbol}(q) \neq a$, is used in an accepting run on w, then q is entered after reading k $\text{symbol}(q)$ symbols, and hence $\text{in}_{\text{symbol}(q)}(q) \geq k$. Contradiction. For rule (1c), again, if q is the accepting state of some run on w, then q must be preceded by k $\text{symbol}(q)$ symbols, and hence $\text{in}_{\text{symbol}(q)}(q) \geq k$, contradiction.

We now show $L(B) \subseteq L(A) \cap \Sigma^{(k)}$. Thereto, let $B = (Q^B, q_0^B, \delta_B, F^B)$. We first make the following observation. Let (q, a, q') be a transition in δ_B. Since B does not contain useless states, it is easy to see that $\text{in}_a(q') \geq \text{in}_a(q) + 1$. As rule (1a) can not be applied to transition (q, a, q'), we now obtain the following equality:

$$\text{For } q, q' \in Q^B, a \in \Sigma, \text{ with } (q, a, q') \in \delta_B : \text{in}_a(q) + 1 = \text{in}_a(q')\,. \tag{4.2}$$

We are now ready to show that $L(B) \subseteq L(A) \cap \Sigma^{(k)}$. Since $L(B) \subseteq L(A)$ definitely holds, it suffices to show that $L(B) = L(B) \cap \Sigma^{(k)}$, i.e., B only accepts strings in $\Sigma^{(k)}$.

Towards a contradiction, suppose that B accepts a string $w \notin \Sigma^{(k)}$. Then, there must exist $i \in [1, k-1]$, $a \in \Sigma$, $u \in L(\varepsilon + \Sigma^*(\Sigma \setminus \{a\}))$ and $v \in L(\varepsilon + (\Sigma \setminus \{a\})\Sigma^*)$, such that $w = ua^iv$. Furthermore, as $w \in L(B)$, there also exist states $p_0, \ldots, p_i$, such that $(q_0, u, p_0) \in \delta_B^*$, $(p_i, v, q_f) \in \delta_B^*$ for some $q_f \in F$, and $(p_j, a, p_{j+1}) \in \delta_B$ for all $j \in [0, i-1]$.

We first argue that $\text{in}_a(p_i) = k$. By (4.1) it suffices to show that $\text{in}_a(p_i) \geq k$. Notice that, as $i > 0$, $\text{symbol}(p_i) = a$. We consider two cases. If $v = \varepsilon$, then $p_i \in F$ and hence by rule (1c) $\text{in}_a(p_i) \geq k$. Otherwise, $v = bv'$ for $b \in \Sigma$, $b \neq a$ and $v' \in \Sigma^*$ and hence p_i must have an outgoing b-labeled transition, with $b \neq a = \text{symbol}(p_i)$. By rule (1b), $\text{in}_a(p_i) \geq k$ must hold.

Now, by repeatedly applying equation (4.2), we obtain $\text{in}_a(p_j) = k - (i - j)$, for all $j \in [0, i]$ and, in particular, $\text{in}_a(p_0) = k - i > 0$. This gives us the desired contradiction. To see why, we distinguish two cases. If $u = \varepsilon$, then $p_0 = q_0$. As A was non-returning and we did not introduce any new transitions, B is also non-returning and hence $\text{in}_a(q_0) = 0$ should hold. Otherwise, if $u = u'b$ for $u' \in \Sigma^*$ and $b \neq a$, then p_0 has an incoming b transition. As B is state-labeled,

p_0 does not have an incoming a transition, and hence, again, $\text{in}_a(p_0) = 0$ should hold. This concludes the proof. $\square$

Now, we are finally ready to prove the desired theorem:

Theorem 39. For every $n \in \mathbb{N}$, there are regular expressions $s_1, \dots, s_m$, with $m = 4n + 3$, each of size $\mathcal{O}(n)$, such that any regular expression defining $L(s_1) \,\&\, L(s_2) \,\&\, \cdots \,\&\, L(s_m)$ is of size at least $2^{\frac{1}{24}(2^n-8)} - 1$.

Proof. Let $n \in \mathbb{N}$, and let $r_1, \dots, r_m$, with $m = 4n + 3$, be the regular expressions obtained in Lemma 36 such that $\bigcap_{i \leq m} L(r_i) = \mathcal{L}_{2^n}$.

Now, similar to Lemma 30, it is shown in [79] that given $r_1, \dots, r_m$ it is possible to construct regular expressions $s_1, \dots, s_m$ such that (1) for all $i \in [1, m]$, $|s_i| \leq 2|r_i|$, and if we define $\mathcal{N}_{2^n} = L(s_1) \,\&\, \cdots \,\&\, L(s_m)$, then (2) $\text{index}(\mathcal{N}_{2^n}) \leq m$, and (3) for every $w \in \Sigma^*$, $w \in \bigcap_{i \leq m} L(r_i)$ if and only if $\text{pump}_m(w) \in \mathcal{N}_{2^n}$. Furthermore, it follows immediately from the construction in [79] that any string in $\mathcal{N}_{2^n} \cap \Sigma^{(m)}$ is of the form $a_1^m c^m a_2^m c^m \cdots a_l^m c^m$, i.e., $\text{pump}_m(w)$ for some $w \in \Sigma^*$.

Since $\bigcap_{i \leq m} L(r_i) = \mathcal{L}_{2^n}$, and $\mathcal{M}_{2^n} = \{\text{pump}_m(w) \mid w \in \mathcal{L}_{2^n}\}$, it hence follows that $\mathcal{M}_{2^n} = \mathcal{N}_{2^n} \cap \Sigma^{(m)}$. As furthermore, by Lemma 37, $\text{sh}(\mathcal{M}_{2^n}) = 2^n$ and $\text{index}(\mathcal{N}_{2^n}) \leq m$, it follows from Lemma 38 that $\text{sh}(\mathcal{N}_{2^n}) \geq \frac{\text{sh}(\mathcal{M}_{2^n})}{|\Sigma|} = \frac{2^n}{8}$. So, $\mathcal{N}_{2^n}$ can be described by the interleaving of the expressions s_1 to s_m, each of size $\mathcal{O}(n)$, but any regular expression defining $\mathcal{N}_{2^n}$ must, by Theorem 22, be of size at least $2^{\frac{1}{24}(2^n-8)} - 1$. This completes the proof. $\square$

Corollary 40. For any $n \in \mathbb{N}$, there exists an RE(&) r_n of size $\mathcal{O}(n^2)$ such that any regular expression defining $L(r_n)$ must be of size at least $2^{\frac{1}{24}(2^n-8)} - 1$.

This completes the chapter. As a final remark, we note that all lower bounds in this chapter make use of a constant size alphabet and can furthermore easily be extended to a binary alphabet. For any language over an alphabet $\Sigma = \{a_1, \dots, a_k\}$, we obtain a new language over the alphabet $\{b, c\}$ by replacing, for any $i \in [1, k]$, every symbol a_i by $b^i c^{k-i+1}$. Obviously, the size of a regular expression for this new language is at most $k + 1$ times the size of the original expression, and the lower bounds on the number of states of DFAs trivially carry over. Furthermore, it is shown in [81] that this transformation does not affect the star height, and hence the lower bounds on the sizes of the regular expression also carry over.

5

Complexity of Extended Regular Expressions

In this chapter, we study the complexity of the equivalence, inclusion, and intersection non-emptiness problem for regular expressions extended with counting and interleaving operators. The main reason for this study is pinpointing the complexity of these decision problems for XML schema languages, as done in Chapter 8. Therefore, we also investigate CHAin Regular Expressions (CHAREs) extended with a counting operator. CHAREs are a subclass of the regular expressions introduced by Martens et. al. [75] relevant to XML schema languages. To study these problems, we introduce NFA($\#, \&$)s, an extension of NFAs with counter and split/merge states for dealing with counting and interleaving operators. An overview of the results is given in Tables 5.1 and 5.2.

Outline. In **Section 5.1**, we define NFA($\#, \&$). In **Section 5.2** we establish the complexity of the basic decision problems for different classes of regular expressions.

5.1 Automata for RE($\#, \&$)

We introduce the automaton model NFA($\#, \&$). In brief, an NFA($\#, \&$) is an NFA with two additional features: *(i)* split and merge transitions to handle interleaving; and, *(ii)* counting states and transitions to deal with numerical

	INCLUSION	EQUIVALENCE	INTERSECTION
RE	PSPACE ([102])	PSPACE ([102])	PSPACE ([70])
RE(&)	EXPSPACE ([79])	EXPSPACE ([79])	**PSPACE**
RE(#)	EXPSPACE ([83])	EXPSPACE ([83])	**PSPACE**
RE(#, &)	**EXPSPACE**	**EXPSPACE**	**PSPACE**
NFA(#), NFA(&), and NFA(#, &)	**EXPSPACE**	**EXPSPACE**	**PSPACE**

Table 5.1: Overview of some new and known complexity results. All results are completeness results. The new results are printed in bold.

occurrence constraints. The idea of split and merge transitions stems from Jędrzejowicz and Szepietowski [60]. Their automata are more general than NFA(#, &) as they can express shuffle-closure which is not regular. Counting states are also used in the counter automata of Kilpeläinen and Tuhkanen [65], and Reuter [93] although these counter automata operate quite differently from NFA(#)s. Zilio and Lugiez [26] also proposed an automaton model that incorporates counting and interleaving by means of Presburger formulas. None of the cited papers consider the complexity of the basic decision problems of their model. We will use NFA(#, &)s to obtain complexity upper bounds in Section 5.2 and Chapter 8.

For readability, we denote $\Sigma \cup \{\varepsilon\}$ by Σ_ε. We then define an NFA(#, &) as follows.

Definition 41. An NFA(#, &) is a tuple $A = (Q, s, f, \delta)$ where

- Q is a finite set of states. To every $q \in Q$, we associate a lower bound $\min(q) \in \mathbb{N}$ and an upper bound $\max(q) \in \mathbb{N} \cup \{\infty\}$, such that $\min(q) \leq \max(q)$;
- $s, f \in Q$ are the start and final states, respectively; and,
- δ is the transition relation and is a subset of the union of the following sets:

(1)	$Q \times \Sigma_\varepsilon \times Q$	ordinary transition (resets the counter)
(2)	$Q \times \{\text{store}\} \times Q$	transition not resetting the counter
(3)	$Q \times \{\text{split}\} \times Q \times Q$	split transition
(4)	$Q \times Q \times \{\text{merge}\} \times Q$	merge transition

Let $\max(A) = \max\{\{\max(q) \mid q \in Q \wedge \max(q) \in \mathbb{N}\} \cup \{\min(q) \mid q \in Q \wedge \max(q) = \infty\}\}$. A *configuration* γ is a pair (P, α) where $P \subseteq Q$ is a set of states and $\alpha : Q \to \{0, \dots, \max(A)\}$ is the value function mapping states to the value of their counter. For a state $q \in Q$, we denote by α_q the value function mapping q to 1 and every other state to 0. The initial configuration γ_s is $(\{s\}, \alpha_s)$. The final configuration γ_f is $(\{f\}, \alpha_f)$. When α is a value function then $\alpha[q = 0]$ denotes the function obtained from α by setting the value of q to 0 while leaving other values unchanged. Similarly, $\alpha[q^{++}]$ increments the value of q by 1, unless $\max(q) = \infty$ and $\alpha(q) = \min(q)$ in which case also the value of q remains unchanged. The rationale behind the latter is that for a state which is allowed an unlimited number of iterations, as $\max(q) = \infty$, it is sufficient to know that $\alpha(q)$ is at least $\min(q)$.

We now define the transition relation between configurations. Intuitively, the value of the state at which the automaton arrives is always incremented by one. When exiting a state, the state's counter is always reset to zero, except when we exit through a *counting transition*, in which case the counter remains the same. In addition, exiting a state through a non-counting transition is only allowed when the value of the counter lies between the allowed minimum and maximum. The latter, hence, ensures that the occurrence constraints are satisfied. *Split* and *merge transitions* start and close a parallel composition.

A configuration $\gamma' = (P', \alpha')$ *immediately follows* a configuration $\gamma = (P, \alpha)$ by reading $\sigma \in \Sigma_\varepsilon$, denoted $\gamma \rightarrow_{A,\sigma} \gamma'$, when one of the following conditions hold:

1. **(ordinary transition)** there is a $q \in P$ and $(q, \sigma, q') \in \delta$ such that $\min(q) \leq \alpha(q) \leq \max(q)$, $P' = (P - \{q\}) \cup \{q'\}$, and $\alpha' = \alpha[q = 0][q'^{++}]$. That is, A is in state q and moves to state q' by reading σ (note that σ can be ε). The latter is only allowed when the counter value of q is between the lower and upper bound. The state q is replaced in P by q'. The counter of q is reset to zero and the counter of q' is incremented by one.

2. **(counting transition)** there is a $q \in P$ and $(q, \text{store}, q') \in \delta$ such that $\alpha(q) < \max(q)$, $P' = (P - \{q\}) \cup \{q'\}$, and $\alpha' = \alpha[q'^{++}]$. That is, A is in state q and moves to state q' by reading ε when the counter of q has not reached its maximal value yet. The state q is replaced in P by q'. The counter of q is not reset but remains the same. The counter of q' is incremented by one.

3. **(split transition)** there is a $q \in P$ and $(q, \text{split}, q_1, q_2) \in \delta$ such that $\min(q) \leq \alpha(q) \leq \max(q)$, $P' = (P - \{q\}) \cup \{q_1, q_2\}$, and $\alpha' = \alpha[q = 0][q_1{}^{++}][q_2{}^{++}]$. That is, A is in state q and splits into states q_1 and q_2 by reading ε when the counter value of q is between the lower and upper bound. The state q in P is replaced by (split into) q_1 and q_2. The counter of q is reset to zero, and the counters of q_1 and q_2 are incremented by

one.

4. **(merge transition)** there are $q_1, q_2 \in P$ and $(q_1, q_2, \text{merge}, q) \in \delta$ such that, for each $j = 1, 2$, $\min(q_j) \leq \alpha(q_j) \leq \max(q_j)$, $P' = (P - \{q_1, q_2\}) \cup \{q\}$, and $\alpha' = \alpha[q_1 = 0][q_2 = 0][q^{++}]$. That is, A is in states q_1 and q_2 and moves to state q by reading ε when the respective counter values of q_1 and q_2 are between the lower and upper bounds. The states q_1 and q_2 in P are replaced by (merged into) q, the counters of q_1 and q_2 are reset to zero, and the counter of q is incremented by one.

For a string w and two configurations γ, γ', we denote by $\gamma \Rightarrow_{A,w} \gamma'$ when there is a sequence of configurations $\gamma \rightarrow_{A,\sigma_1} \cdots \rightarrow_{A,\sigma_n} \gamma'$ such that $w = \sigma_1 \cdots \sigma_n$. The latter sequence is called a *run* when γ is the initial configuration γ_s. A string w is *accepted* by A if and only if $\gamma_s \Rightarrow_{A,w} \gamma_f$ with γ_f the final configuration. We usually denote $\Rightarrow_{A,w}$ simply by $\Rightarrow_w$ when A is clear from the context. We denote by $L(A)$ the set of strings accepted by A. The size of A, denoted by $|A|$, is $|Q| + |\delta| + \Sigma_{q \in Q} \log(\min(q)) + \Sigma_{q \in Q} \log(\max(q))$. So, each $\min(q)$ and $\max(q)$ is represented in binary.

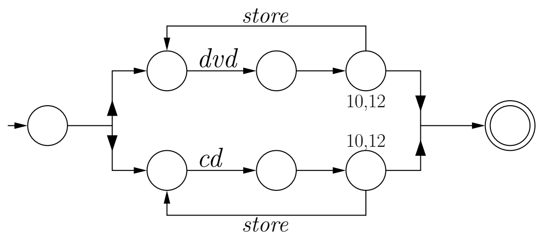


Figure 5.1: An NFA(#, &) for the language *dvd*^{10,12} & *cd*^{10,12}. For readability, we only displayed the alphabet symbol on non-epsilon transitions and counters for states q where $\min(q)$ and $\max(q)$ are different from one. The arrows from the initial state and to the final state are split and merge transitions, respectively. The arrows labeled *store* represent counting transitions.

An example of an NFA(#, &) defining *dvd*^{10,12} & *cd*^{10,12} is shown in Figure 5.1. An NFA(#) is an NFA(#, &) without split and merge transitions. An NFA(&) is an NFA(#, &) without counting transitions.

Clearly NFA(#, &) accept all regular languages. The next theorem shows the complexity of translating between RE(#, &) and NFA(#, &), and between NFA(#, &) and NFA.

In its proof we will make use of the CORRIDOR TILING problem. Thereto, a *tiling instance* is a tuple $T = (X, H, V, \overline{b}, \overline{t}, n)$ where X is a finite set of tiles, $H, V \subseteq X \times X$ are the horizontal and vertical constraints, n is an integer in unary notation, and $\bar{b}, \bar{t}$ are n-tuples of tiles ($\bar{b}$ and $\bar{t}$ stand for *bottom row* and *top row*, respectively).

A *correct corridor tiling for* T is a mapping $\lambda : \{1, \ldots, m\} \times \{1, \ldots, n\} \to X$ for some $m \in \mathbb{N}$ such that the following constraints are satisfied:

- the bottom row is $\bar{b}$: $\bar{b} = (\lambda(1,1), \ldots, \lambda(1,n))$;
- the top row is $\bar{t}$: $\bar{t} = (\lambda(m,1), \ldots, \lambda(m,n))$;
- all vertical constraints are satisfied: $\forall i < m$, $\forall j \leq n$, $(\lambda(i,j), \lambda(i+1,j)) \in V$; and,
- all horizontal constraints are satisfied: $\forall i \leq m$, $\forall j < n$, $(\lambda(i,j), \lambda(i,j+1)) \in H$.

The CORRIDOR TILING problem asks, given a tiling instance, whether there exists a correct corridor tiling. The latter problem is PSPACE-complete [104]. We are now ready to prove Theorem 42.

Theorem 42. (1) Given an RE($\#, \&$) expression r, an equivalent NFA($\#, \&$) can be constructed in time linear in the size of r.

(2) Given an NFA($\#, \&$) A, an equivalent NFA can be constructed in time exponential in the size of A.

Proof. (1) We prove the theorem by induction on the structure of RE($\#, \&$)-expressions. For every r we define a corresponding NFA($\#, \&$) $A(r) = (Q_r, s_r, f_r, \delta_r)$ such that $L(r) = L(A(r))$.

For r of the form ε, a, $r_1 \cdot r_2$, $r_1 + r_2$ and r_1^* these are the usual RE to NFA constructions with ε-transition as displayed in text books such as [55].

We perform the following steps for the numerical occurrence and interleaving operator which are graphically illustrated in Figure 5.2. The construction for the interleaving operator comes from [60].

(i) If $r = (r_1)^{k,\ell}$ and $A(r_1) = (Q_1, s_1, f_1, \delta_1)$, then

- $Q_r := Q_{r_1} \uplus \{s_r, f_r, q_r\}$;
- $\min(s_r) = \max(s_r) = \min(f_r) = \max(f_r) = 1$, $\min(q_r) = k$, and $\max(q_r) = \ell$;
- if $k \neq 0$ then $\delta_r := \delta_{r_1} \uplus \{(s_r, \varepsilon, s_{r_1}), (f_{r_1}, \varepsilon, q_r), (q_r, \text{store}, s_{r_1}), (q_r, \varepsilon, f_r)\}$; and,
- if $k = 0$ then $\delta_r := \delta_{r_1} \uplus \{(s_r, \varepsilon, s_{r_1}), (f_{r_1}, \varepsilon, q_r), (q_r, \text{store}, s_{r_1}), (q_r, \varepsilon, f_r), (s_r, \varepsilon, f_r)\}$.

(ii) If $r = r_1 \mathbin{\&} r_2$, $A(r_1) = (Q_{r_1}, s_{r_1}, f_{r_1}, \delta_{r_1})$ and $A(r_2) = (Q_{r_2}, s_{r_2}, f_{r_2}, \delta_{r_2})$, then

- $Q_r := Q_{r_1} \uplus Q_{r_2} \uplus \{s_r, f_r\}$;
- $\min(s_r) = \max(s_r) = \min(f_r) = \max(f_r) = 1$;
- $\delta_r := \delta_{r_1} \uplus \delta_{r_2} \uplus \{(s_r, \text{split}, s_{r_1}, s_{r_2}), (f_{r_1}, f_{r_2}, \text{merge}, f_r)\}$.

Notice that in each step of the construction, a constant number of states are added to the automaton. Moreover, the constructed counters are linear in the size of r. It follows that the size of $A(r)$ is linear in the size of r. The correctness of the construction can easily be proved by induction on the structure of r.

(2) Let $A = (Q_A, s_A, f_A, \delta_A)$ be an NFA(#, &). We define an NFA $B = (Q_B, q_B^0, F_B, \delta_B)$ such that $L(A) = L(B)$. Formally,

- $Q_B = 2^{Q_A} \times (\{1, \ldots, \max(A)\}^{Q_A})$;
- $s_B = (\{s_A\}, \alpha_{s_A})$;
- $F_B = \{(\{f_A\}, \alpha_{f_A})\}$ if $\varepsilon \notin L(A)$, and $F_B = \{(\{f_A\}, \alpha_{f_A}), (\{s_A\}, \alpha_{s_A})\}$, otherwise;
- $\delta_B = \{\big((P_1, \alpha_1), \sigma, (P_2, \alpha_2)\big) \mid \sigma \in \Sigma$ and $(P_1, \alpha_1) \Rightarrow_{A,\sigma} (P_2, \alpha_2)$ for configurations (P_1, α_1) and (P_2, α_2) of $A\}$.

Obviously, B can be constructed from A in exponential time. Notice that the size of Q_B is smaller than $2^{|Q_A|} \cdot 2^{|A| \cdot |Q_A|}$. Furthermore, as B mimics A by storing the configurations of B in its states, it is immediate that $L(A) = L(B)$. □

We next turn to the complexity of the basic decision problems for NFA(#, &).

Theorem 43. (1) EQUIVALENCE and INCLUSION for NFA(#, &) are EXPSPACE-complete;

(2) INTERSECTION for NFA(#, &) is PSPACE-complete; and,

(3) MEMBERSHIP for NFA(#) is NP-hard, MEMBERSHIP for NFA(&) and NFA(#, &) are PSPACE-complete.

Proof. (1) EXPSPACE-hardness follows from Theorem 42(1) and the EXPSPACE-hardness of EQUIVALENCE for RE(&) [79]. Membership in EXPSPACE follows from Theorem 42(2) and the fact that INCLUSION for NFAs is in PSPACE [102].

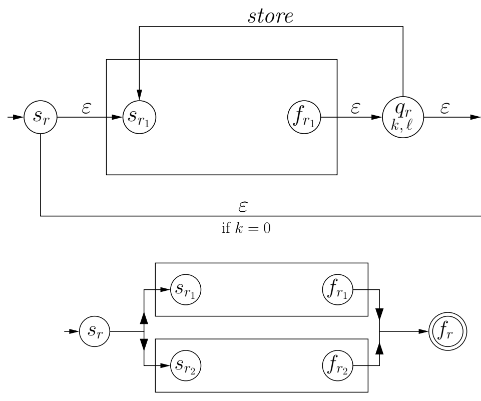


Figure 5.2: From RE(#, &) to NFA(#, &).

(2) The lower bound follows from [70]. We show that the problem is in PSPACE. For $j \in \{1, \ldots, n\}$, let $A_j = (Q_j, s_j, f_j, \delta_j)$ be an NFA(#, &). The algorithm proceeds by guessing a Σ-string w such that $w \in \bigcap_{j=1}^{n} L(A_j)$. Instead of guessing w at once, we guess it symbol by symbol and keep for each A_j one current configuration γ_j on the tape. More precisely, at each time instant, the tape contains for each A_j a configuration $\gamma_j = (P_j, \alpha_j)$ such that $\gamma_{s_j} \Rightarrow_{A_j, w_i} (P_j, \alpha_j)$, where $w_i = a_1 \cdots a_i$ is the prefix of w guessed up to now. The algorithm accepts when each γ_j is a final configuration. Formally, the algorithm operates as follows.

1. Set $\gamma_j = (\{s_j\}, \alpha_{s_j})$ for $j \in \{1, \ldots, n\}$;
2. While not every γ_j is a final configuration
   - (i) Guess an $a \in \Sigma$.
   - (ii) Non-deterministically replace each γ_j by a (P'_j, α'_j) such that $(P_j, \alpha_j) \Rightarrow_{A_j, a} (P'_j, \alpha'_j)$.

As the algorithm only uses space polynomial in the size of the NFA(#, &) and step (2,ii) can be done using only polynomial space, the overall algorithm operates in PSPACE.

(3) The MEMBERSHIP problem for NFA(#, &)s is easily seen to be in PSPACE by an on-the-fly implementation of the construction in Theorem 42(2). Indeed, as a configuration of an NFA(#, &) $A = (Q, s, f, \delta)$ has size at most $|Q| + |Q| \cdot \log(\max(A))$, we can store a configuration using only polynomial space.

We show that the MEMBERSHIP problem for NFA(#)s is NP-hard by a reduction from a modification of INTEGER KNAPSACK. We define this problem as follows. Given a set of natural numbers $W = \{w_1, \ldots, w_k\}$ and two integers

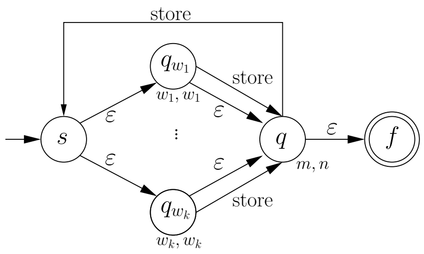


Figure 5.3: NP-hardness of MEMBERSHIP for NFA(#).

m and n, all in binary notation, the problem asks whether there exists a mapping $\tau : W \to \mathbb{N}$ such that $m \leq \sum_{w \in W} \tau(w) \times w \leq n$. The latter mapping is called a solution. This problem is known to be NP-complete [36].

We construct an NFA(#) $A = (Q, s, f, \delta)$ such that $L(A) = \{\varepsilon\}$ if W, m, n has a solution, and $L(A) = \emptyset$ otherwise.

The set of states Q consists of the start and final states s and f, a state q_{w_i} for each weight w_i, and a state q. Intuitively, a successful computation of A loops at least m and at most n times through state q. In each iteration, A also visits one of the states q_{w_i}. Using numerical occurrence constraints, we can ensure that a computation accepts if and only if it passes at least m and at most n times through q and a multiple of w_i times through each q_{w_i}. Hence, an accepting computation exists if and only if there is a mapping τ such that $m \leq \sum_{w \in W} \tau(w) \times w \leq n$.

Formally, the transitions of A are the following:

- $(s, \varepsilon, q_{w_i})$ for each $i \in \{1, \ldots, k\}$;
- $(q_{w_i}, \text{store}, q)$ for each $i \in \{1, \ldots, k\}$;
- $(q_{w_i}, \varepsilon, q)$ for each $i \in \{1, \ldots, k\}$;
- (q, store, s); and,
- (q, ε, f).

We set $\min(s) = \max(s) = \min(f) = \max(f) = 1$, $\min(q) = m$, $max(q) = n$ and $\min(q_{w_i}) = \max(q_{w_i}) = w_i$ for each q_{w_i}. The automaton is graphically illustrated in Figure 5.3.

Finally, we show that MEMBERSHIP for NFA(&)s is PSPACE-hard. Before giving the proof, we describe some n-ary merge and split transitions which can be rewritten in function of the regular binary split and merge transitions.

1. $(q_1, q_2, \text{merge-split}, q_1', q_2')$: States q_1 and q_2 are read, and immediately split into states q_1' and q_2'.

2. $(q_1, q_2, q_3, \text{merge-split}, q_1', q_2', q_3')$: States q_1, q_2 and q_3 are read, and immediately split into states q_1', q_2' and q_3'.
3. $(q_1, \text{split}, q_1', \dots, q_n')$: State q_1 is read, and is immediately split into states $q_1', \dots, q_n'$.
4. $(q_1, \dots, q_n, \text{merge}, q_1')$: States $q_1, \dots, q_n$ are read, and are merged into state q_1'.

Transitions of type 1 (resp. 2) can be rewritten using 2 (resp. 4) *regular* transitions, and 1 (resp. 3) new auxiliary states. Transitions of type 3 and 4 can be rewritten using $(n-1)$ regular transitions and $(n-1)$ new auxiliary states. For example, the transition $(q_1, q_2, \text{merge-split}, q_1', q_2')$ is equal to the transitions $(q_1, q_2, \text{merge}, q_h)$, and $(q_h, \text{split}, q_1', q_2')$, where q_h is a new auxiliary state.

To show that MEMBERSHIP for NFA(&)s is PSPACE-hard, we reduce from CORRIDOR TILING. Given a tiling instance $T = (X, H, V, \bar{b}, \bar{t}, n)$, we construct an NFA(&) A over the empty alphabet ($\Sigma = \emptyset$) which accepts ε if and only if there exists a correct corridor tiling for T.

The automaton constructs the tiling row by row. Therefore, A must at any time reflect the current row in its state set. (recall that an NFA(&) can be in more than one state at once) To do this, A contains for every tile x a set of states $x^1, \dots, x^n$, where n is the length of each row. If A is in state x^i, this means that the ith tile of the current row is x. For example, if $\bar{b} = x_1x_3x_1$, and $\bar{t} = x_2x_2x_1$, then the initial state set is $\{x_1^1, x_3^2, x_1^3\}$, and A can accept when the state set is $\{x_2^1, x_2^2, x_1^3\}$.

It remains to describe how A can transform the current row ("state set"), into a state set which describes a valid row on top of the current row. This transformation proceeds on a tile by tile basis and begins with the first tile, say x_i, in the current row which is represented by x_i^1 in the state set. Now, for every tile x_j, for which $(x_i, x_j) \in V$, we allow x_i^1 to be replaced by x_j^1, since x_j can be the first tile of the row on top of the current row. For the second tile of the next row, we have to replace the second tile of the current row, say x_k, by a new tile, say x_ℓ, such that the vertical constraints between x_k and x_ℓ are satisfied and such that the horizontal constraints between x_ℓ and the tile we just placed at the first position of the first row, x_j, are satisfied as well.

The automaton proceeds in this manner for the remainder of the row. For this, the automaton needs to know at any time at which position a tile must be updated. Therefore, an extra set of states $p_1, \dots, p_n$ is created, where the state p_i says that the tile at position i has to be updated. So, the state set always consists of one state p_i, and a number of states which represent the current and next row. Here, the states up to position i already represent the tiles of the next row, the states from position i still represent the current row, and i is the next position to be updated.

We can now formally construct an NFA(&) $A = (Q, s, f, \delta)$ which accepts ε if and only if there exists a correct corridor tiling for a tiling instance $T = (X, H, V, \bar{b}, \bar{t}, n)$ as follows:

- $Q = \{x^j \mid x \in X, 1 \leq j \leq n\} \cup \{p_i \mid 1 \leq i \leq n\} \cup \{s, f\}$
- $\Sigma = \emptyset$
- δ is the union of the following transitions
  - $(s, \text{split}, p_1, \bar{b}_1^1, \ldots, \bar{b}_n^n)$: From the initial state the automaton immediately goes to the states which represent the bottom row.
  - $(p_1, \bar{t}_1^1, \ldots, \bar{t}_n^n, \text{merge}, f)$: When the state set represents a full row (the automaton is in state p_1), and the current row is the accepting row, all states are merged to the accepting state.
  - $\forall x_i, x_j \in X, (x_j, x_i) \in V$: $(p_1, x_j^1, \text{merge-split}, p_2, x_i^1)$: When the first tile has to be updated, the automaton only has to check the vertical constraints with the first tile of the previous row.
  - $\forall x_i, x_j, x_k \in X, m \in \mathbb{N}, 2 \leq m \leq n, (x_k, x_i) \in V, (x_j, x_i) \in H$: $(p_m, x_k^m, x_j^{m-1}, \text{merge-split}, p_{(m \bmod n)+1}, x_i^m, x_j^{m-1})$: When a tile at the mth $(m \neq 1)$ position has to be updated, the automaton has to check the vertical constraint with the mth tile at the previous row, and the horizontal constraint with the $(m-1)$th tile of the new row.

Clearly, if there exists a correct corridor tiling for T, there exists a run of A accepting ε. Conversely, the construction of our automaton, in which the updates are always determined by the position p_i, and the horizontal and vertical constraints, assures that when there is an accepting run of A on ε, this run simulates a correct corridor tiling for T. $\square$

5.2 Complexity of Regular Expressions

We now investigate the complexity of regular expressions, extended with counting and interleaving, and CHAREs.

Mayer and Stockmeyer [79] and Meyer and Stockmeyer [83] already established the EXPSPACE-completeness of INCLUSION and EQUIVALENCE for RE(&) and RE(#), respectively. From Theorem 42(1) and Theorem 43(1) it then directly follows that allowing both operators does not increase the complexity. It

	INCLUSION	EQUIVALENCE	INTERSECTION
CHARE	PSPACE [75]	in PSPACE [102]	PSPACE [75]
CHARE(#)	**EXPSPACE**	**in EXPSPACE**	**PSPACE**
CHARE($a, a?$)	coNP [75]	in PTIME [75]	NP [75]
CHARE(a, a^*)	coNP [75]	in PTIME [75]	NP [75]
CHARE($a, a?, a\#$)	**coNP**	**in PTIME**	**NP**
CHARE($a, a\#^{>0}$)	**in PTIME**	**in PTIME**	**in PTIME**

Table 5.2: Overview of new and known complexity results concerning Chain Regular Expressions. All results are completeness results, unless mentioned otherwise. The new results are printed in bold.

further follows from Theorem 42(1) and Theorem 43(2) that INTERSECTION for RE(#, &) is in PSPACE. We stress that the latter results could also have been obtained without making use of NFA(#, &)s but by translating RE(#, &)s directly to NFAs. However, in the case of INTERSECTION such a construction should be done in an on-the-fly fashion to not go beyond PSPACE. Although such an approach certainly is possible, we prefer the shorter and more elegant construction using NFA(#, &)s.

Theorem 44. 1. EQUIVALENCE and INCLUSION for RE(#, &) are in EXPSPACE; and

2. INTERSECTION for RE(#, &) is PSPACE-complete.

Proof. (1) Follows directly from Theorem 42(1) and Theorem 43(1).

(2) The upper bound follows directly from Theorem 42(1) and Theorem 43(2). The lower bound is already known for ordinary regular expressions. □

Bex et al. [10] established that the far majority of regular expressions occurring in practical DTDs and XSDs are of a very restricted form as defined next. The class of *chain regular expressions (CHAREs)* are those REs consisting of a sequence of factors $f_1 \cdots f_n$ where every factor is an expression of the form $(a_1 + \cdots + a_n)$, $(a_1 + \cdots + a_n)?$, $(a_1 + \cdots + a_n)^+$, or, $(a_1 + \cdots + a_n)^*$, where $n \geq 1$ and every a_i is an alphabet symbol. For instance, the expression $a(b+c)^*d^+(e+f)?$ is a CHARE, while $(ab+c)^*$ and $(a^* + b?)^*$ are not.[1]

We introduce some additional notation to define subclasses and extensions of CHAREs. By CHARE(#) we denote the class of CHAREs where also factors of the form $(a_1 + \cdots + a_n)^{k,\ell}$, with $k, \ell \in \mathbb{N} \cup \{0\}$ are allowed. For

[1]We disregard here the additional restriction used in [9] that every symbol can occur only once.

the following fragments, we list the admissible types of factors. Here, a, $a?$, a^* denote the factors $(a_1 + \cdots + a_n)$, $(a_1 + \cdots + a_n)?$, and $(a_1 + \cdots + a_n)^*$, respectively, with $n = 1$, while $a\#$ denotes $a^{k,\ell}$, and $a\#^{>0}$ denotes $a^{k,\ell}$ with $k > 0$.

Table 5.2 lists the new and the relevant known results. We first show that adding numerical occurrence constraints to CHAREs increases the complexity of INCLUSION by one exponential. We reduce from EXP-CORRIDOR TILING.

Theorem 45. INCLUSION for CHARE(#) is EXPSPACE-complete.

Proof. The EXPSPACE upper bound already follows from Theorem 44(1).

The proof for the EXPSPACE lower bound is similar to the proof for PSPACE-hardness of INCLUSION for CHAREs in [75]. The main difference is that the numerical occurrence operator allows to compare tiles over a distance exponential in the size of the tiling instance.

The proof is a reduction from EXP-CORRIDOR TILING. A *tiling instance* is a tuple $T = (X, H, V, x_\perp, x_\top, n)$ where X is a finite set of tiles, $H, V \subseteq X \times X$ are the horizontal and vertical constraints, $x_\perp, x_\top \in X$, and n is a natural number in unary notation. A *correct exponential corridor tiling for T* is a mapping $\lambda : \{1, \ldots, m\} \times \{1, \ldots, 2^n\} \to X$ for some $m \in \mathbb{N}$ such that the following constraints are satisfied:

- the first tile of the first row is $x_\perp$: $\lambda(1, 1) = x_\perp$;
- the first tile of the m-th row is $x_\top$: $\lambda(m, 1) = x_\top$;
- all vertical constraints are satisfied: $\forall i < m$, $\forall j \leq 2^n$, $(\lambda(i, j), \lambda(i + 1, j)) \in V$; and,
- all horizontal constraints are satisfied: $\forall i \leq m$, $\forall j < 2^n$, $(\lambda(i, j), \lambda(i, j + 1)) \in H$.

The EXP-CORRIDOR TILING problem asks, given a tiling instance, whether there exists a correct exponential corridor tiling. The latter problem is easily shown to be EXPSPACE-complete [104].

We proceed with the reduction from EXP-CORRIDOR TILING. Thereto, let $T = (X, H, V, x_\perp, x_\top, n)$ be a tiling instance. Without loss of generality, we assume that $n \geq 2$. We construct two CHARE(#) expressions r_1 and r_2 such that

$$L(r_1) \subseteq L(r_2) \text{ if and only if } \text{there exists no correct exponential corridor tiling for } T.$$

As EXPSPACE is closed under complement, the EXPSPACE-hardness of INCLUSION for CHARE(#) follows.

Set $\Sigma = X \uplus \{\$, \triangle\}$. For ease of exposition, we denote $X \cup \{\triangle\}$ by $X_\triangle$ and $X \cup \{\triangle, \$\}$ by $X_{\triangle,\$}$. We encode candidates for a correct tiling by a string in which the rows are separated by the symbol $\triangle$, that is, by strings of the form

$$\triangle R_0 \triangle R_1 \triangle \cdots \triangle R_m \triangle, \qquad (\dagger)$$

in which each R_i represents a row, that is, belongs to X^{2^n}. Moreover, R_0 is the bottom row and R_m is the top row. The following regular expressions detect strings of this form which do not encode a correct tiling for T:

- $X_\triangle^* \triangle X^{0,2^n-1} \triangle X_\triangle^*$. This expression detects rows that are too short, that is, contain less than 2^n symbols.
- $X_\triangle^* \triangle X^{2^n+1,2^n+1} X^* \triangle X_\triangle^*$. This expression detects rows that are too long, that is, contain more than 2^n symbols.
- $X_\triangle^* x_1 x_2 X_\triangle^*$, for every $x_1, x_2 \in X$, $(x_1, x_2) \notin H$. These expressions detect all violations of horizontal constraints.
- $X_\triangle^* x_1 X_\triangle^{2^n,2^n} x_2 X_\triangle^*$, for every $x_1, x_2 \in X$, $(x_1, x_2) \notin V$. These expressions detect all violations of vertical constraints.

Let $e_1, \ldots, e_k$ be an enumeration of the above expressions. Notice that $k = \mathcal{O}(|X|^2)$. It is straightforward that a string w in (†) does not match $\bigcup_{i=1}^{k} e_i$ if and only if w encodes a correct tiling.

Let $e = e_1 \cdots e_k$. Because of leading and trailing $X_\triangle^*$ expressions, $L(e) \subseteq L(e_i)$, for every $i \in \{1, \ldots, k\}$. We are now ready to define r_1 and r_2:

$$\begin{array}{rcl} r_1 & = & \overbrace{\$e\$e\$\cdots\$e\$}^{k \text{ times } e} \triangle x_\perp X^{2^n-1,2^n-1} \triangle X_\triangle^* \triangle x_\top X^{2^n-1,2^n-1} \triangle \overbrace{\$e\$e\$\cdots\$e\$}^{k \text{ times } e} \\ r_2 & = & \$X_{\triangle,\$}^* \$e_1\$e_2\$\cdots\$e_k\$X_{\triangle,\$}^*\$ \end{array}$$

Notice that both r_1 and r_2 are in CHARE(#) and can be constructed in polynomial time. It remains to show that $L(r_1) \subseteq L(r_2)$ if and only if there is no correct tiling for T.

We first show the implication from left to right. Thereto, let $L(r_1) \subseteq L(r_2)$. Let uwu' be an arbitrary string in $L(r_1)$ such that $u, u' \in L(\$e\$e\$\cdots\$e\$)$ and $w \in \triangle x_\perp X^{2^n-1,2^n-1} \triangle X_\triangle^* \triangle x_\top X^{2^n-1,2^n-1} \triangle$. By assumption, $uwu' \in L(r_2)$.

Notice that uwu' contains $2k+2$ times the symbol "\$". Moreover, the first and the last "\$" of uwu' is always matched onto the first and last "\$" of r_2. This means that $k+1$ consecutive \$-symbols of the remaining $2k$ \$-symbols in uwu' must be matched onto the \$-symbols in $\$e_1\$e_2\$\cdots\$e_k\$$.

Hence, w is matched onto some e_i. So, w does not encode a correct tiling. As the subexpression $\triangle x_\perp X^{2^n-1,2^n-1} \triangle X^*_\triangle \triangle x_\top X^{2^n-1,2^n-1} \triangle$ of r_1 defines all candidate tilings, the system T has no solution.

To show the implication from right to left, assume that there is a string $uwu' \in L(r_1)$ that is not in r_2, where $u, u' \in L(\$e\$e\$ \cdots \$e\$)$. Then $w \notin \bigcup_{i=1}^k L(e_i)$ and, hence, w encodes a correct tiling. □

Adding numerical occurrence constraints to the fragment CHARE$(a, a?)$ keeps EQUIVALENCE in PTIME, INTERSECTION in NP and INCLUSION in coNP.

Theorem 46. (1) EQUIVALENCE for CHARE$(a, a?, a\#)$ is in PTIME.

(2) INCLUSION for CHARE$(a, a?, a\#)$ is coNP-complete.[2]

(3) INTERSECTION for CHARE$(a, a?, a\#)$ is NP-complete.

Proof. (1) It is shown in [75] that two CHARE$(a, a?)$ expressions are equivalent if and only if they have the same *sequence normal form* (which is defined below). As $a^{k,\ell}$ is equivalent to $a^k(a?)^{\ell-k}$, it follows that two CHARE$(a, a?, a\#)$ expressions are equivalent if and only if they have the same sequence normal form. It remains to argue that the sequence normal form of CHARE$(a, a?, a\#)$ expressions can be computed in polynomial time. To this end, let $r = f_1 \cdots f_n$ be a CHARE$(a, a?, a\#)$ expression with factors $f_1, \ldots, f_n$. The *sequence normal form* is then obtained in the following way. First, we replace every factor of the form

- a by $a[1, 1]$;
- $a?$ by $a[0, 1]$; and,
- $a^{k,\ell}$ by $a[k, \ell]$,

where a is an alphabet symbol. We call a the *base symbol* of the factor $a[i, j]$. Then, we replace successive subexpressions $a[i_1, j_1]$ and $a[i_2, j_2]$ carrying the same base symbol a by $a[i_1 + i_2, j_1 + j_2]$ until no further replacements can be made anymore. For instance, the sequence normal form of $aa?a^{2,5}a?bb?b?b^{1,7}$ is $a[3, 8]b[2, 10]$. Obviously, the above algorithm to compute the sequence normal form of CHARE$(a, a?, a\#)$ expressions can be implemented in polynomial time. It can then be tested in linear time whether two sequence normal forms are the same.

(2) coNP-hardness is immediate since INCLUSION is already coNP-complete for CHARE$(a, a?)$ expressions [75].

[2]In the extended abstract presented at ICDT'07, in which these results first appeared, the complexity was wrongly attributed to lie between PSPACE and EXPSPACE.

We show that the problem remains in coNP. To this end, we represent strings w by their sequence normal form as discussed above, where we take each string w as the regular expression defining w. We call such strings *compressed.* Let r_1 and r_2 be two CHARE$(a, a?, a\#)$s. We can assume that they are in sequence normal form.

To show that $L(r_1) \not\subseteq L(r_2)$, we guess a compressed string w of polynomial size for which $w \in L(r_1)$, but $w \notin L(r_2)$. We guess $w \in L(r_1)$ in the following manner. We iterate from left to right over the factors of r_1. For each factor $a[k, \ell]$ we guess an h such that $k \leq h \leq \ell$, and add a^h to the compressed string w. This algorithm gives a compressed string of polynomial size which is defined by r_1. Furthermore, this algorithm is capable of guessing every possible string defined by r_1. It is however possible that in the compressed string there are two consecutive *elements* a^i, a^j with the same *base symbol* a. If this is the case we merge these elements to a^{i+j} which gives a proper compressed string.

The following lemma shows that testing $w \notin L(r_2)$ can be done in polynomial time.

Lemma 47. Given a compressed string w and an expression r in sequence normal form, deciding whether $w \in L(r)$ is in PTIME.

Proof. Let $w = a_1^{p_1} \cdots a_n^{p_n}$, and $r = b_1[k_1, \ell_1] \cdots b_m[k_m, \ell_m]$. Denote $b_i[k_i, \ell_i]$ by f_i. For every position i of w $(0 < i \leq n)$, we define C_i as a set of factors $b[k, \ell]$ of r. Formally, $f_j \in C_i$ when $a_1^{p_1} \cdots a_{i-1}^{p_{i-1}} \in L(f_1 \cdots f_{j-1})$ and $a_i = b_j$. We compute the C_i as follows.

- C_1 is the set of all $b_j[k_j, \ell_j]$ such that $a_1 = b_j$, and $\forall h < j, k_h = 0$. These are all the factors of r which can match the first symbol of w.

- Then, for all $i \in \{2, \ldots, n\}$, we compute C_i from C_{i-1}. In particular, $f_h = b_h[k_h, \ell_h] \in C_i$ when there is a $f_j = b_j[k_j, \ell_j] \in C_{i-1}$ such that $a_{i-1}^{p_{i-1}} \in f_j \cdots f_{h-1}$ and $a_i = b_h$. That is, the following conditions should hold:

  - $j < h$: f_h occurs after f_j in r.
  - $b_h = a_i$: f_h can match the first symbol of $a_i^{p_i}$.
  - $\forall e \in \{j, \ldots, h-1\}$, if $b_e \neq a_{i-1}$ then $k_e = 0$: in between factors f_j and f_h it is possible to match only symbols a_{i-1}.
  - Let min $= \sum_{e\in\{j,\ldots,h-1\}, b_e = a_{i-1}} k_e$ and max $= \sum_{e\in\{j,\ldots,h-1\}, b_e = a_{i-1}} \ell_e$. Then min $\leq p_{i-1} \leq$ max. That is, p_{i-1} symbols a_{i-1} should be matched from f_j to f_{h-1}.

Then, $w \in L(r)$ if and only if there is an $f_j \in C_n$ such that $a_n^{p_n} \in L(f_j \cdots f_n)$. As the latter test and the computation of $C_1, \ldots, C_n$ can be done in polynomial time, the lemma follows. □

(3) NP-hardness is immediate since INTERSECTION is already NP-complete for CHARE$(a, a?)$ expressions [75].

We show that the problem remains in NP. As in the proof of Theorem 46(2) we represent a string w as a compressed string. Let $r_1, \ldots, r_n$ be CHARE$(a, a?, a\#)$ expressions.

Lemma 48. If $\bigcap_{i=1}^n L(r_i) \neq \emptyset$, then there exists a string $w = a_1^{p_1} \cdots a_m^{p_m} \in \bigcap_{i=1}^n L(r_i)$ such that $m \leq \min\{|r_i| \mid i \in \{1, \ldots, n\}\}$ and, for each $i \in \{1, \ldots, n\}$, j_i is not larger than the largest integer occurring in $r_1, \ldots, r_n$.

Proof. Suppose that there exists a string $w = a_1^{p_1} \cdots a_m^{p_m} \in \bigcap_{i=1}^n L(r_i)$, with $a_i \neq a_{i+1}$ for every $i \in \{1, \ldots, m-1\}$. Since w is matched by every expression $r_1, \ldots, r_n$, and since a factor of a CHARE$(a, a?, a\#)$ expression can never match a strict superstring of $a_i^{p_i}$ for $i \in \{1, \ldots, n\}$, we have that $m \leq \min\{|r_i| \mid i \in \{1, \ldots, n\}\}$.

Furthermore, since w is matched by every expression $r_1, \ldots, r_n$, no j_i can be larger than the largest integer occurring in $r_1, \ldots, r_n$. □

The NP algorithm then consists of guessing a compressed string w of polynomial size and verifying whether $w \in \bigcap_{i=1}^n L(r_i)$. If we represent $r_1, \ldots, r_n$ by their sequence normal form, this verification step can be done in polynomial time by Lemma 47. □

Finally, we exhibit a tractable subclass with numerical occurrence constraints:

Theorem 49. INCLUSION, EQUIVALENCE, and INTERSECTION for CHARE$(a, a\#^{>0})$ are in PTIME.

Proof. The upper bound for EQUIVALENCE is immediate from Theorem 46(2).

For INCLUSION, let r_1 and r_2 be two CHARE$(a, a\#^{>0})$s in sequence normal form. (as defined in the proof of Theorem 46) Let $r_1 = a_1[k_1, \ell_1] \cdots a_n[k_n, \ell_n]$ and $r_2 = a'_1[k'_1, \ell'_1] \cdots a'_{n'}[k'_{n'}, \ell'_{n'}]$. Notice that every number k_i and k'_j is greater than zero. We claim that $L(r_1) \subseteq L(r_2)$ if and only if $n = n'$ and for every $i \in \{1, \ldots, n\}$, $a_i = a'_i$, $k_i \geq k'_i$, and $\ell_i \leq \ell'_i$.

Indeed, if $n \neq n'$, or if there exists an i such that $a_i \neq a'_i$ or $k_i < k'_i$, then $a_1^{k_1} \cdots a_n^{k_n} \in L(r_1) \setminus L(r_2)$. If there exists an i such that $\ell_i > \ell'_i$, then $a_1^{\ell_1} \cdots a_n^{\ell_n} \in L(r_1) \setminus L(r_2)$. Conversely, it is immediate that every string in

$L(r_1)$ is also in $L(r_2)$. It is straightforward to test these conditions in linear time.

For INTERSECTION, let, for every $i \in \{1,\ldots,n\}$, $r_i = a_{i,1}[k_{i,1},\ell_{i,1}]\cdots a_{i,m_i}[k_{i,m_i},\ell_{i,m_i}]$ be a CHARE($a, a\#^{>0}$) in sequence normal form. Notice that every number $k_{i,j}$ is greater than zero. We claim that $\bigcap_{i=1}^{n} L(r_i) \neq \emptyset$ if and only if

(i) $m_1 = m_2 = \cdots = m_n$;

(ii) for every $i, j \in \{1,\ldots,n\}$ and $x \in \{1,\ldots,m_1\}$, $a_{i,x} = a_{j,x}$; and,

(iii) for every $x \in \{1,\ldots,m_1\}$, $\max\{k_{i,x} \mid 1 \leq i \leq n\} \leq \min\{\ell_{i,x} \mid 1 \leq i \leq n\}$.

Indeed, if the above conditions hold, we have that $a_{1,1}^{K_1}\cdots a_{1,m_1}^{K_{m_1}}$ is in $\bigcap_{i=1}^{n} L(r_i)$, where $K_x = \max\{k_{i,x} \mid 1 \leq i \leq n\}$ for every $x \in \{1,\ldots,m_1\}$. If $m_i \neq m_j$ for some $i, j \in \{1,\ldots,n\}$, then the intersection between r_i and r_j is empty. So assume that condition (i) holds. If $a_{i,x} \neq a_{j,x}$ for some $i, j \in \{1,\ldots,n\}$ and $x \in \{1,\ldots,m_1\}$, then we also have that the intersection between r_i and r_j is empty. Finally, if condition (iii) does not hold, take i, j, and x such that $k_{i,x} = \max\{k_{i,x} \mid 1 \leq i \leq n\}$ and $\ell_{j,x} = \min\{\ell_{i,x} \mid 1 \leq i \leq n\}$. Then the intersection between r_i and r_j is empty.

Finally, testing conditions (i)–(iii) can be done in linear time. $\square$

6 Deterministic Regular Expressions with Counting

As mentioned before, XML Schema expands the vocabulary of regular expression by a counting operator, but restricts them by requiring the expressions to be deterministic. Although there is a notion of determinism which has become the standard in the context of standard regular expressions, there are in fact two slightly different notions of determinism. The most common notion, weak determinism (also called one-unambiguity [17]), intuitively requires that, when matching a string from left to right against an expression, it is always clear against which position in the expression the next symbol in the string must be matched. For example, the expression $(a + b)^*a$ is not weakly deterministic, because it is not clear in advance to which a in the expression the first a of the string aaa must be matched. The reason is that, without looking ahead, we do not know whether the current a we read is the last symbol of the string or not. On the other hand, the equivalent expression $b^*a(b^*a)^*$ is weakly deterministic: the first a we encounter must be matched against the leftmost a in the expression, and all other a's against the rightmost one (similarly for the b's). Strong determinism restricts regular expressions even further. Intuitively, it 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.

Although the latter example illustrates the difference between the notions of weak and strong determinism, they in fact almost coincide for standard regular expressions. Indeed, Brüggemann-Klein [15] has shown that any weak deterministic expression can be translated into an equivalent strongly deterministic one in linear time.[1] However, this situation changes completely when counting is involved. First, the algorithm for deciding whether an expression is weakly deterministic is non-trivial [66]. For instance, $(a^{2,3} + b)^{2,2}b$ is weakly deterministic, but the very similar $(a^{2,3} + b)^{3,3}b$ 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 chapter is an in-depth study of the notions of weak and strong determinism in the presence of counting with respect to expressiveness, succinctness, and complexity.

We first 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 regular languages are definable by weakly deterministic expressions with counting.

Then, 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 [15].

We also 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. Bruggemann-Klein [15] 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. Combining the results on CDFAs with the latter result then also allows to infer better upper bounds on the inclusion and equivalence problem

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

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 [66].

It should be said that XML Schema uses weakly deterministic expressions with counting. However, it is also noted by Sperberg-McQueen [101], 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 [65], Zilio and Lugiez [26], and Sperberg-McQueen [101]. Here, Sperberg-McQueen introduces the extension of the Glushkov construction which we study in Section 6.5. We introduce a new automata model in Section 6.4 as none of these models allow to derive all results in Sections 6.4 and 6.5. Further, Sperberg-McQueen [101] and Koch and Scherzinger [68] introduce a (slightly different) notion of strongly deterministic expression with and without counting, respectively. We follow the semantical meaning of Sperberg-McQueen's definition, while using the technical approach of Koch and Scherzinger. Finally, Kilpelainen [63] shows that inclusion for weakly deterministic expressions with counting is coNP-hard; and Colazzo, Ghelli, and Sartiani [24] have investigated the inclusion problem involving subclasses of deterministic expressions with counting.

Concerning deterministic languages without counting, the seminal paper is by Bruggemann-Klein and Wood [17] 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 [40, 37, 64, 83].

Outline. In **Section 6.1**, we provide some additional definitions. In **Section 6.2** we study the expressive power of weak and strong deterministic expressions with counting, and in **Section 6.3** their relative succinctness. In **Section 6.4**, we define CNFA, and in **Section 6.5** we study the Glushkov construction translating RE(#) to CNFA. In **Sections 6.6** and **6.7** we consider the complexity of testing whether an expression with counting is strongly deterministic and decision problems for deterministic RE(#). We conclude in **Section 6.8**.

6.1 Preliminaries

We introduce some additional notation concerning (deterministic) RE(#) expressions. An RE(#) expression r is *nullable* if $\varepsilon \in L(r)$. We say that an RE(#) r is in *normal form* if for every nullable subexpression $s^{k,l}$ of r we have $k = 0$. Any RE(#) can easily be normalized in linear time. Therefore, we assume that all expressions used in this chapter are in normal form. Sometimes we will use the following observation, which follows directly from the definitions:

Remark 50. A subexpression $r^{k,\ell}$ is nullable if and only if $k = 0$.

For a regular expression r, we say that a subexpression of r of the form $s^{k,\ell}$ is an *iterator of* r. Let $\overline{r}$ and $\overline{s}$ be expressions such that $\overline{s}$ is a subexpression of $\overline{r}$. Then we say that $\overline{s}$ is a *factor* of $\overline{r}$, whenever $\text{first}(\overline{s}) \subseteq \text{first}(\overline{r})$ and $\text{last}(\overline{s}) \subseteq \text{last}(\overline{r})$. When $\overline{r}$ and $\overline{s}$ are iterators, this intuitively means that a number of iterations of $\overline{s}$ are sufficient to satisfy $\overline{r}$. For instance, in the expression $\overline{s} = (a_1^{0,3} b_1^{1,2})^{3,4}$, $b_1^{1,2}$ is a factor of $\overline{s}$, but $a_1^{0,3}$ is not.

For two symbols $\overline{x}$, $\overline{y}$ of a marked expression $\overline{r}$, we denote by $\text{lca}(\overline{x}, \overline{y})$ the smallest subexpression of $\overline{r}$ containing both $\overline{x}$ and $\overline{y}$. Further, we write $\overline{s} \preceq \overline{r}$ when $\overline{s}$ is a subexpression of $\overline{r}$ and $\overline{s} \prec \overline{r}$ when $\overline{s} \preceq \overline{r}$ and $\overline{s} \neq \overline{r}$. For an iterator $s^{k,\ell}$, let $\text{base}(s^{k,\ell}) := s$, $\text{lower}(s^{k,\ell}) := k$, and $\text{upper}(s^{k,\ell}) := \ell$. We say that $s^{k,\ell}$ is *bounded* when $\ell \in \mathbb{N}$, otherwise it is *unbounded*.

Weak determinism. For completeness, we define here the notion of weak determinism. This notion, however, is exactly the notion of determinism as defined in Section 2.2. Recall that for an expression r, $\overline{r}$ denotes a marking of r, and $\text{Char}(\overline{r})$ denotes the symbols occurring in $\overline{r}$.

Definition 51. An RE(#) expression r is *weakly deterministic* if, for all strings $\overline{u}, \overline{v}, \overline{w} \in \text{Char}(\overline{r})^*$ and all symbols $\overline{a}, \overline{b} \in \text{Char}(\overline{r})$, 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(#) expression. The classes of all weakly deterministic languages with counting, respectively, without counting, are denoted by $\text{DRE}^{\#}_{W}$, respectively, DRE_{W}.

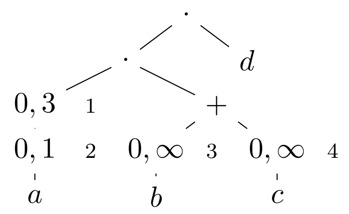


Figure 6.1: Parse tree of $(a^{0,1})^{0,3}(b^{0,\infty} + c^{0,\infty})d$. Counter nodes are numbered from 1 to 4.

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 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 $((a + \varepsilon)(b + \varepsilon))^{1,2}$ 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 [101], while using the formal approach of Koch and Scherzinger [68] (who called the notion *strong one-unambiguity*)[2]. We denote the *parse tree* of an RE(#) expression r by pt(r). Figure 6.1 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

[2]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^*.

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, $([_1([_2 a]_2)^{0,1}]_1)^{0,3}(([_3 b]_3)^{0,\infty} + ([_4 c]_4)^{0,\infty})d$ 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 6.1. 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 52. 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 $\alpha \neq \beta$ over Γ, and a symbol $a \in \Sigma$ such that $u\alpha a v$ and $u\beta a w$ 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 $\mathrm{DRE}^{\#}_S$, respectively, DRE_S, denotes all languages definable by a strongly deterministic expressions with, respectively, without, counting.

6.2 Expressive Power

Brüggemann-Klein and Wood [17] proved that, for any alphabet Σ, DRE_W forms a strict subclass of the regular languages, denoted $\mathrm{REG}(\Sigma)$. The complete picture of the relative expressive power of the different classes depends on the size of Σ, as shown in Figure 6.2.

Figure 6.2: An overview of the expressive power of different classes of deterministic regular languages, depending on the alphabet size. Numbers refer to the theorems proving the (in)equalities.

$$\mathrm{DRE}_S \overset{[15]}{=} \mathrm{DRE}_W \overset{54}{=} \mathrm{DRE}^{\#}_S \overset{55}{=} \mathrm{DRE}^{\#}_W \overset{60}{\subsetneq} \mathrm{REG}(\Sigma) \quad (\text{if } |\Sigma| = 1)$$

$$\mathrm{DRE}_S \overset{[15]}{=} \mathrm{DRE}_W \overset{54}{=} \mathrm{DRE}^{\#}_S \overset{59}{\subsetneq} \mathrm{DRE}^{\#}_W \overset{60}{\subsetneq} \mathrm{REG}(\Sigma) \quad (\text{if } |\Sigma| \geq 2)$$

The equality $\mathrm{DRE}_S = \mathrm{DRE}_W$ is already implicit in the work of Brüggemann-Klein [15].[3] By this result and by definition, all inclusions from left to right

[3]Strong determinism was not explicitly considered in [15]. However, the paper gives a transformation of weakly deterministic regular expressions into equivalent expressions in *star normal form*; and every expression in star normal form is strongly deterministic.

already hold. The rest of this section is devoted to proving the other equalities and inequilaties in Theorems 54, 55, 59, and 60, while the intermediate lemmas are only used in proving the former theorems.

We need a small lemma to prepare for the equality between DRE_S and $\mathrm{DRE}_S^{\#}$.

Lemma 53. Let r_1r_2 be a factor of $r^{k,\ell}$ with r_1, r_2 nullable, $L(r_1) \neq \{\varepsilon\}$, $L(r_2) \neq \{\varepsilon\}$, and $\ell \geq 2$. Then $r^{k,\ell}$ is not strongly deterministic.

Proof. If $r^{k,\ell}$ is not weakly deterministic, the lemma already holds. Therefore, assume that $r^{k,\ell}$ is weakly deterministic. Let $\overline{r}^{k,\ell}$ be a marked version of $r^{k,\ell}$. Let $\widetilde{\overline{s}} := [_1\widetilde{\overline{r}}]_1^{k,\ell}$ denote the bracketed version of $\overline{r}^{k,\ell}$. Denote by $\widetilde{\overline{r}}_1$ and $\widetilde{\overline{r}}_2$ the subexpressions of $\widetilde{\overline{s}}$ that correspond to $\overline{r}_1$ and $\overline{r}_2$.

Let $\widetilde{\overline{u}}$ (respectively, $\widetilde{\overline{v}}$) be a non-empty string in $L(\widetilde{\overline{r}}_1)$ (respectively, $L(\widetilde{\overline{r}}_2)$). These strings exist as $L(r_1)$ and $L(r_2)$ are not equal to $\{\varepsilon\}$. Let $\widetilde{\overline{w}} = [_1\widetilde{\overline{u}}\widetilde{\overline{v}}]_1$. Define $x := \max\{0, k-2\}$. Then, both $[_1\widetilde{\overline{u}}]_1[_1\widetilde{\overline{v}}]_1\widetilde{\overline{w}}^x$ and $[_1\widetilde{\overline{u}}\widetilde{\overline{v}}]_1\widetilde{\overline{w}}^{x+1}$ are in $L(\widetilde{\overline{s}})$ and thereby show that $r^{k,\ell}$ is not strongly deterministic. $\square$

Theorem 54. Let Σ be an arbitrary alphabet. Then,

$$\mathrm{DRE}_S = \mathrm{DRE}_S^{\#}\,.$$

Proof. Let r be an arbitrary strongly deterministic regular expression with counting. We can assume without loss of generality that no subexpressions of the form $s^{1,1}$ or ε occur. Indeed, iteratively replacing any subexpression of the form $s^{1,1}$, $s\varepsilon$ or εs by s; $s+\varepsilon$ or $\varepsilon+s$ by $s^{0,1}$ and $\varepsilon^{k,\ell}$ by ε, yields an expression which is still strongly deterministic and which is either equal to ε or does not contain $s^{1,1}$ or ε. As in the former case we are done, we can hence assume the latter.

We recursively transform r into a strongly deterministic expression $\mathrm{EC}(r)$ without counting such that $L(r) = L(\mathrm{EC}(r))$, using the rules below. In these rules, EC stands for "eliminate counter" and ECE for "eliminate counter and epsilon":

(a) for all $a \in \Sigma$, $\mathrm{EC}(a) := a$

(b) $\mathrm{EC}(r_1 + r_2) := \mathrm{EC}(r_1) + \mathrm{EC}(r_2)$

(c) $\mathrm{EC}(r_1r_2) := \mathrm{EC}(r_1)\mathrm{EC}(r_2)$

(d) if r nullable then $\mathrm{EC}(r^{0,1}) := \mathrm{EC}(r)$

(e) if r not nullable then $\mathrm{EC}(r^{0,1}) := \mathrm{ECE}(r) + \varepsilon$

(f) $\mathrm{EC}(r^{k,\infty}) := \mathrm{ECE}(r) \cdots \mathrm{ECE}(r) \cdot \mathrm{ECE}(r)^*$

(g) if $\ell \in \mathbb{N}\backslash\{1\}$ then $\mathrm{EC}(r^{k,\ell}) := \mathrm{ECE}(r)\cdots\mathrm{ECE}(r)\cdot\big(\mathrm{ECE}(r)(\mathrm{ECE}(r)(\cdots)+\varepsilon)+\varepsilon\big)$

In the last two rules, $\mathrm{ECE}(r)\cdots\mathrm{ECE}(r)$ denotes a k-fold concatenation of $\mathrm{ECE}(r)$, and the recursion in $\big(\mathrm{ECE}(r)(\mathrm{ECE}(r)(\cdots)+\varepsilon)+\varepsilon\big)$ contains $\ell - k$ occurrences of r.

(a') for all $a \in \Sigma$, $\mathrm{ECE}(a) := a$

(b') $\mathrm{ECE}(r_1 + r_2) := \mathrm{ECE}(r_1) + \mathrm{ECE}(r_2)$

(c') $\mathrm{ECE}(r_1 r_2) := \mathrm{EC}(r_1)\mathrm{EC}(r_2)$

(d') if $k \neq 0$, then $\mathrm{ECE}(r^{k,\ell}) := \mathrm{EC}(r^{k,\ell})$

(e') $\mathrm{ECE}(r^{0,1}) := \mathrm{ECE}(r)$

(f') $\mathrm{ECE}(r^{0,\infty}) := \mathrm{ECE}(r)\mathrm{ECE}(r)^*$

(g') if $k = 0$ and $\ell \in \mathbb{N} \setminus \{1\}$, then $\mathrm{ECE}(r^{k,\ell}) := \mathrm{ECE}(r)\big(\mathrm{ECE}(r)(\cdots)+\varepsilon\big)$

Similarly as above, the recursion $\big(\mathrm{ECE}(r)(\cdots)+\varepsilon\big)$ contains $\ell-1$ occurrences of r. We prove that $L(r) = L(\mathrm{EC}(r))$ and that $\mathrm{EC}(r)$ is a strongly deterministic regular expression.

To show $L(r) = L(\mathrm{EC}(r))$, we prove by simultaneous induction on the application of the rules (a)–(g) and (a')–(g') that $L(\mathrm{EC}(r)) = L(r)$, for any strongly deterministic expression r, and that $L(\mathrm{ECE}(r)) = L(r) \setminus \{\varepsilon\}$, for all expressions r to which $\mathrm{ECE}(r)$ can be applied. The latter is important as $\mathrm{ECE}(r)$ does not equal $L(r) \setminus \{\varepsilon\}$ for all r. For instance, $\mathrm{ECE}(a^{0,1}b^{0,1}) = (a+\varepsilon)(b+\varepsilon)$, and $L(a^{0,1}b^{0,1}) = L((a+\varepsilon)(b+\varepsilon))$. However, as EC is always applied to a strongly deterministic expression, we will see that ECE will never be applied to such a subexpression. This is why we need Lemma 53.

The base cases of the induction are (a) $\mathrm{EC}(a) := a$ and (a') $\mathrm{ECE}(a) := a$. These cases are clear. For the induction, the cases (b)–(e) are trivial. Correctness for cases (f) and (g) is immediate from Remark 50. Case (b') is also trivial.

The first non-trivial case is (c'). If $\varepsilon \notin L(r_1 r_2)$ then (c') is clearly correct. Towards a contradiction, suppose that $\varepsilon \in L(r_1 r_2)$. This implies that r_1 and r_2 are nullable. Notice that we only apply rule (c') when rewriting r if either *(i)* $r_1 r_2$ is a factor of some $s^{0,1}$ with s not nullable (case (e)), or *(ii)* $r_1 r_2$ is a factor of a subexpression of the form $s^{k,\ell}$ with $\ell \geq 2$ (cases (f,g)). Here, $r_1 r_2$ must in both cases be a factor since, whenever ECE is applied to a subexpression, this subexpression is a factor; and the factor relation is clearly transitive. The reason that there must always be such a superexpression to which case (e), (f), or (g) is applied is that the recursive process starts by applying EC. Therefore, we must have applied (e), (f), or (g) to some superexpression of which r_1r_2 is a factor before applying ECE to r_1r_2.

In case *(i)*, r_1 and r_2 nullable implies that r in rule (e) must be nullable (because r_1r_2 is a factor of r), which contradicts the precondition of rule (e). In case *(ii)*, Lemma 53 claims that $s^{k,\ell}$ and therefore r is not strongly deterministic, which is also a contradiction.

By Remark 50, (d') is also correct. Cases (e')–(g') are also trivial. This concludes the proof that $L(\mathrm{EC}(r)) = L(r)$.

We next prove that $\mathrm{EC}(r)$ and $\mathrm{ECE}(r)$ are strongly deterministic regular expression, whenever r is strongly deterministic. We prove this by induction on the reversed order of application of the rewrite rules. The induction base rules are (a) and (a'), which are immediate. Furtermore, rules (b)–(e) and (b')–(e') are also immediate. For instance, for rule (b) we know by induction that $\mathrm{EC}(r_1)$ and $\mathrm{EC}(r_2)$ are strongly deterministic. As $L(r_1) = L(\mathrm{EC}(r_1))$, $L(r_2) = L(\mathrm{EC}(r_2))$, and $r_1 + r_2$ is strongly deterministic, it follows that also $\mathrm{EC}(r_1) + \mathrm{EC}(r_2)$ is strongly deterministic.

Cases (f), (g), (f'), and (g') are more involved. We only investigate case (f) because these four cases are all very similar. Let $\overline{\mathrm{EC}(r^{k,\infty})}$ denote a marked version of $\mathrm{EC}(r^{k,\infty})$ and let $\overline{\mathrm{EC}(r^{k,\infty})} = \overline{\mathrm{ECE}(r)}_1 \cdots \overline{\mathrm{ECE}(r)}_k \overline{\mathrm{ECE}(r)}^*_{k+1}$. Further, let $\overline{r}^{k,\infty}$ be a marking of $r^{k,\infty}$ and let $\mathrm{f} : \mathrm{Char}(\overline{\mathrm{EC}(r^{k,\infty})}) \to \mathrm{Char}(\overline{r}^{k,\infty})$ be the natural mapping associating each symbol in $\overline{\mathrm{EC}(r^{k,\infty})}$ to its corresponding symbol in $\mathrm{Char}(\overline{r}^{k,\infty})$. For instance, when $r = (aba)^{1,\infty}$, then $\mathrm{EC}(r) = (aba)(aba)^*$, $\overline{r} = (a_1b_1a_2)^{1,\infty}$, $\overline{\mathrm{EC}(r)} = (a_1b_1a_2)(a_3b_2a_4)^*$, and $\mathrm{f}(a_1) = a_1$, $\mathrm{f}(b_1) = b_1$, $\mathrm{f}(a_2) = a_2$, $\mathrm{f}(a_3) = a_1$, $\mathrm{f}(b_2) = b_1$, and $\mathrm{f}(a_4) = a_2$. By abuse of notation, we also let f denote its natural extension mapping strings to strings.

Now, assume, towards a contradiction, that $\mathrm{EC}(r^{k,\infty})$ is not strongly deterministic, and assume first that this is due to the fact that $\mathrm{EC}(r^{k,\infty})$ is not weakly deterministic. Hence, there exist marked strings $\overline{u}, \overline{v}, \overline{w}$ and marked symbols $\overline{x}$ and $\overline{y}$ such that $\overline{uxv}$ and $\overline{uyw}$ in $L(\overline{\mathrm{EC}(r^{k,\infty})})$ with $\overline{x} \neq \overline{y}$ and $\mathrm{dm}(\overline{x}) = \mathrm{dm}(\overline{y})$. We distinguish two cases. First, assume $\mathrm{f}(\overline{x}) \neq \mathrm{f}(\overline{y})$. Then, as both $\mathrm{f}(\overline{u})\mathrm{f}(\overline{x})\mathrm{f}(\overline{v})$ and $\mathrm{f}(\overline{u})\mathrm{f}(\overline{x})\mathrm{f}(\overline{w})$ are in $L(\overline{r}^{k,\infty})$ we immediately obtain a contradiction with the weak determinism, and thus also the strong determinism, of $r^{k,\infty}$. Second, assume $\mathrm{f}(\overline{x}) = \mathrm{f}(\overline{y})$. Then, $\overline{x}$ and $\overline{y}$ cannot occur in the same $\overline{\mathrm{ECE}(r)}_i$, for any $i \in [1, k+1]$. Indeed, otherwise $\mathrm{ECE}(r)$ would not be weakly deterministic, contradicting the induction hypothesis. Hence we can assume $\overline{x} \in \mathrm{Char}(\overline{\mathrm{ECE}(r)}_i)$ and $\overline{y} \in \mathrm{Char}(\overline{\mathrm{ECE}(r)}_j)$, with $1 \leq i < j \leq k+1$. But then, consider the strings $w_1 = \mathrm{dm}(\overline{uxv})$ and $w_2 = \mathrm{dm}(\overline{uyw})$ and notice that $\mathrm{dm}(\overline{ux}) = \mathrm{dm}(\overline{uy})$. Let $[_m$ be the opening bracket corresponding to the iterator $\overline{r}^{k,\infty}$ in the bracketed version of $\overline{r}^{k,\infty}$. Then, we can construct two correctly bracketed words defined by the bracketing of $\overline{r}^{k,\infty}$ by adding brackets to w_1 and w_2 such that (i) in the prefix $\mathrm{dm}(\overline{ux})$ of w_1, i $[_m$-brackets occur (indicating i iterations of the outermost iterator) and (ii) in the prefix $\mathrm{dm}(\overline{uy})$ of w_2, j $[_m$-brackets occur. But, as $\mathrm{dm}(\overline{ux}) = \mathrm{dm}(\overline{uy})$ this implies that $\overline{r}^{k,\infty}$ is not strongly deterministic, a contradiction.

Hence, if $\mathrm{EC}(r^{k,\infty})$ is not strongly deterministic, this is due to the failure of the second reason in Definition 52. It is easily seen that, as this reason concerns the iterators, and all subexpressions are strongly deterministic due to the induction hypothesis, it must be the case that $\mathrm{ECE}(r)^*$ is not strongly deterministic. Let $[_{m_1}$ be the opening bracket corresponding to the iterator $\mathrm{ECE}(r)^*$. Since $\mathrm{ECE}(r)$ is strongly deterministic but $\mathrm{ECE}(r)^*$ is not, we can take strings $\widetilde{\overline{u}}_1\alpha_1\widetilde{\overline{av_1}}$ and $\widetilde{\overline{u}}_1\beta_1\widetilde{\overline{aw_1}}$, so that

- $\widetilde{\overline{u}}_1\alpha_1\widetilde{\overline{av_1}}$ and $\widetilde{\overline{u}}_1\beta_1\widetilde{\overline{aw_1}}$ are correctly bracketed and accepted by the bracketed version of $\overline{\mathrm{ECE}(r)}^*$,
- $\alpha_1, \beta_1 \in \Gamma^*$, and
- $]_{m_1}[_{m_1}$ is a substring of α_1 but not of β_1,

Consider the expression $r^{0,\infty}$ (which has a lower bound 0 instead of k) and let $[_{m_2}$ be the opening bracket corresponding to its outermost iterator. According to the above, there also exist correctly bracketed strings $\widetilde{\overline{u}}_2\alpha_2\widetilde{\overline{av_2}}$ and $\widetilde{\overline{u}}_2\beta_2\widetilde{\overline{aw_2}}$ accepted by the bracketed version of $r^{0,\infty}$ such that $]_{m_2}[_{m_2}$ is a substring of α_2 but not of β_2. This proves that $r^{0,\infty}$ and therefore $r^{k,\infty}$ is not strongly deterministic. This hence leads to the desired contradiction, and shows that $\mathrm{EC}(r)$ is indeed strongly deterministic. □

Theorem 55. Let Σ be an alphabet with $|\Sigma| = 1$. Then,

$$\mathrm{DRE}^{\#}_{S} = \mathrm{DRE}^{\#}_{W}$$

Proof. By definition, every strongly deterministic expression is also weakly deterministic. Hence, it suffices to show that, over a unary alphabet, every weakly deterministic language with counting can also be defined by a strongly deterministic expression with counting. We will do so by characterizing the weakly deterministic languages with counting over a unary alphabet in terms of their corresponding minimal DFAs. Thereto, we first introduce some notation. The following notions come from, e.g., Shallit [91], but we repeat them here for completeness. (Shallit used *tail* to refer to what we call a *chain*.)

Definition 56. We say that a DFA over $\Sigma = \{a\}$ is a *chain* if its start state is q_0 and its transition function is of the form $\delta(q_0, a) = q_1, \ldots, \delta(q_{n-1}, a) = q_n$, where $q_i \neq q_j$ when $i \neq j$. A DFA is a *chain followed by a cycle* if its transition function is of the form $\delta(q_0, a) = q_1$, ..., $\delta(q_{n-1}, a) = q_n$, $\delta(q_n, a) = q_{n+1}$, ..., $\delta(q_{n+m-1}, a) = q_{n+m}$, $\delta(q_{n+m}, a) = q_n$, where $q_i \neq q_j$ when $i \neq j$. The *cycle states* of the latter DFA are $q_n, \ldots, q_{n+m}$.

We say that a unary regular language L is *ultimately periodic* if L is infinite and its minimal DFA is a chain followed by a cycle, for which at most one of the cycle states is final.

The crux of the proof then lies in Lemma 57. It is well known (see, e.g., [91]) 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, or a chain followed by a cycle. To this fact, the following lemma adds that for languages defined by weakly deterministic regular expressions only one node in this cycle can be final.

Lemma 57. Let $\Sigma = \{a\}$, and $L \in \mathrm{REG}(\Sigma)$, then $L \in \mathrm{DRE}^{\#}_{W}$ if and only if L is either finite or ultimately periodic.

Before we prove this lemma, note that it implies Theorem 55. Indeed, any finite language can clearly be defined by a strongly deterministic expression, while an infinite but ultimately periodic language can be defined by a strongly deterministic expression of the form $a^{n_1}(a^{n_2}(\cdots a^{n_{k-1}}(a^{n_k})^* \cdots + \varepsilon) + \varepsilon)$, where a^{n_i} denotes the n_i-fold concatenation of a. Hence, it only remains to prove Lemma 57. Thereto, we first need a more refined notion of ultimate periodicity and an additional lemma, after which we conclude with the proof of Lemma 57.

We say that L over alphabet $\{a\}$ is (n_0, x)*-periodic* if

(i) $L \subseteq L((a^x)^*)$, and

(ii) for every $n \in \mathbb{N}$ such that $nx \geq n_0$, L contains the string a^{nx}, i.e., the string of a's of length nx.

We say that L is *ultimately x-periodic* if L is (n_0, x)-periodic for some $n_0 \in \mathbb{N}$. Notice that these notions imply that L is infinite. Clearly, any ultimately x-periodic language is also ultimately periodic. However, the opposite does not always hold. Indeed, in an ultimately x-periodic language *all* strings have lengths which are multiples of x, i.e., they have length 0 (modulo x). In an ultimately periodic language only all *sufficiently long* string must have the same length y (modulo x), for a fixed y, which, moreover, can be different from 0.

Lemma 58. Let L be a language, $x \in \mathbb{N}$, $k \in \mathbb{N} \cup \{0\}$, and $\ell \in \mathbb{N} \cup \{\infty\}$ with $k \leq \ell$. If L is ultimately x-periodic, then $L^{k,\ell}$ is also ultimately x-periodic.

Proof. Every string in $L^{k,\ell}$ is a concatenation of (possibly empty) strings in L. Since the length of every string in L is a multiple of x, it follows that the length of every string in $L^{k,\ell}$ is a multiple of x. Therefore, $L^{k,\ell} \subseteq L((a^x)^*)$.

Furthermore, take $n_0 \in \mathbb{N}$ such that L is (n_0, x)-periodic. Let $k_0 := \max\{k, 1\}$. We will show that $L^{k,\ell}$ is $(k_0(n_0 + x), x)$-periodic, which proves the lemma. Take $n \in \mathbb{N}$ such that $nx \geq k_0(n_0 + x)$. Hence, $(\frac{n}{k_0} - 1)x \geq n_0$ since $k_0 \geq 1$ and therefore $\lfloor \frac{n}{k_0} \rfloor x \geq n_0$. Hence, $a^{nx} = a^{\lfloor n/k_0 \rfloor x} \cdots a^{\lfloor n/k_0 \rfloor x} a^{r_0 x}$, where the dots abbreviate a k_0-fold concatenation of $a^{\lfloor n/k_0 \rfloor x}$ and $r_0 := n \bmod \lfloor n/k_0 \rfloor$. Since L is (n_0, x)-periodic, $a^{\lfloor n/k_0 \rfloor x} \in L$ and $a^{(\lfloor n/k_0 \rfloor + r_0)x} \in L$. Hence, $a^{nx} \in L^{k_0}$, which implies that $a^{nx} \in L^{k,\ell}$. □

We are now finally ready to prove Lemma 57 and thus conclude the proof of Theorem 55
Proof. [of Lemma 57] Clearly, any finite language is in $\text{DRE}^{\#}_{W}$ and any ultimately periodic language can already be defined by a strongly deterministic expression, and hence is also in $\text{DRE}^{\#}_{W}$. Hence, we only need to show that, if an infinite language is in $\text{DRE}^{\#}_{W}$, then it is ultimately periodic, i.e., can be defined with a DFA of the correct form. It is well-known that, for every infinite unary language, the minimal DFA is a chain followed by a cycle. We therefore only need to argue that, in this cycle, at most one of the nodes is final.

Let $\Sigma = \{a\}$ and let r be a weakly deterministic regular expression with counting over $\{a\}$, such that $L(r)$ is infinite. We can assume without loss of generality that r does not contain concatenations with ε. Since r is over a unary alphabet, we can make the following observations:

$$\text{If } r \text{ has a disjunction } r_1 + r_2 \text{ then either } L(r_1) = \{\varepsilon\} \text{ or } L(r_2) = \{\varepsilon\}. \quad (6.1)$$
$$\text{There are no subexpressions of the form } r_1 r_2 \text{ in } r \text{ in which } |L(r_1)| > 1. \quad (6.2)$$

Indeed, if these statements do not hold, then r is not weakly deterministic. Our first goal is to bring r into a normal form. More specifically, we want to write r as

$$r = (r_1(r_2(\cdots(r_n)^{p_n,1}\cdots)^{p_3,1})^{p_2,1})^{p_1,1},$$

where

(a) for each $i = 1, \ldots, n$, $p_i \in \{0, 1\}$;

(b) r has no occurrences of ε;

(c) r_n is a tower of counters, that is, it only has a single occurrence of a; and

(d) $L(r_1), \ldots, L(r_{n-1})$ are singletons, and $L(r_n)$ is infinite.

Notice that, if r has the normal form and one of $L(r_i)$, with $1 \leq i \leq n-1$, is not a singleton, then this would immediately violate (6.2). In order to achieve this normal form, we iteratively replace

(i) all subexpressions of the form $(s^{k,k})^{\ell,\ell}$ with $s^{k\ell,k\ell}$;

(ii) all subexpressions of the form $s^{k_1,k_1}s^{k_2,\ell_2}$ with $s^{k_1+k_2,k_1+\ell_2}$; and

(iii) all subexpressions of the form $(s+\varepsilon)$ and $(\varepsilon+s)$ with $s^{0,1}$

(iv) all subexpressions of the form $a^{k,k}s$, where $s \neq a^{k_1,\ell_1}$ with $a^{k,k}s^{1,1}$

until no such subexpressions occur anymore. These replacements preserve the language defined by r and preserve weak determinism. Due to (6.1) and (6.2), these replacements turn r in the normal form above adhering to the syntactic constraints (a)–(c). Furthermore, we know that condition (d) must hold because r is weakly deterministic.

Hence, we can assume that $r = (r_1(r_2(\cdots(r_n)^{p_n,1}\cdots)^{p_3,1})^{p_2,1})^{p_1,1}$ in which only $L(r_n)$ is infinite. Notice that we can translate the regular expression

$$r' = (r_1(r_2(\cdots(r_{n-1}(X)^{p_n,1})^{p_{n-1},1}\cdots)^{p_3,1})^{p_2,1})^{p_1,1}$$

over alphabet $\{a, X\}$ into a DFA which is a chain and which reads the symbol X precisely once, at the end of the chain. Therefore, it suffices to prove now that we can translate r_n into a DFA which is a chain followed by a cycle, in which at most one of the cycle nodes is final. Indeed, if A_1 and A_2 are the DFAs for r' and r_n respectively, then we can intuitively obtain the DFA A for r by concatenating these two DFAs. More formally, if q_1 is the unique state in A_1 which has the transition $\delta(q_1, X) = q_2$ (and q_2 is final in A_1), and q_3 is the initial state of A_2, then we can obtain A by taking the union of the states and transitions of A_1 and A_2, removing state q_2, and merging states q_1 and q_3 into a new state, while preserving incoming and outgoing transitions. The initial state of A is the initial state of A_1 and its final state set is the union of the final state sets in A_1 and A_2. Since A_1 is a chain, $L(A) = L(r)$ is ultimately periodic if and only if $L(A_2) = L(r_n)$ is ultimately periodic. It thus only remains to show that $L(r_n)$ is ultimately periodic.

Let $s^{k,\infty}$ be the smallest subexpression of r_n in which the upper bound is unbounded. (Such an expression must exist, since $L(r_n)$ is infinite.) We will first prove that $L(s^{k,\infty})$ is ultimately x-periodic for some $x \in \mathbb{N}$. Due to Lemma 58 and the structure of r_n, this also proves that $L(r_n)$ is ultimately x-periodic, and thus ultimately periodic, and concludes our proof.

It therefore only remains to show that $L(s^{k,\infty})$ is ultimately x-periodic for some $x \in \mathbb{N}$. To this end, there are two possibilities: either s contains a subexpression of the form $(s')^{y_1,y_2}$ with $y_1 < y_2$, or it does not. If not, then $L(s^{k,\infty})$ is of the form $(a^{x,x})^{k,\infty}$ due to the replacement (i) above in our normalization. In this case, $L(s^{k,\infty})$ is clearly (xk, x)-periodic.

If $s^{k,\infty}$ contains a subexpression of the form $(s')^{y_1,y_2}$ with $y_1 < y_2$, then it is of the form $(((a^{x,x})^{y_1,y_2})^{h_1^1,h_2^1})\cdots)^{h_1^n,h_2^n})^{k,\infty}$, where x can equal 1, and the h_1^i, h_2^i denote a nesting of iterators. It is immediate that $L(s^{k,\infty}) \subseteq L((a^{x,x})^*)$, i.e., the length of every string in $L(s^{k,\infty})$ is a multiple of x. To show that there also exists an n_0 such that $s^{k,\infty}$ defines *all* strings of length mx with $m \in \mathbb{N}$ and $mx \geq n_0$, let $z = h_1^1 \cdot h_1^2 \cdots h_1^n$, or $z = 1$ when $n = 0$. Clearly, $(((a^{x,x})^{y,y+1})^{z,z})^{k,\infty}$ defines a subset of $s^{k,\infty}$ and, hence, it suffices to show that $(((a^{x,x})^{y,y+1})^{z,z})^{k,\infty}$ defines all such strings of length $mx \geq n_0$.

Let $n_0 = y^2xkz$. We show that for any m such that $mx \geq y^2xkz$, a^{mx} is defined by $(((a^{x,x})^{y,y+1})^{z,z})^{k,\infty}$. Take any $m \geq y^2kz$, and note that it suffices to show that a^m is defined by $((a^{y,y+1})^{z,z})^{k,\infty}$. The latter is the case if there exists an $\ell \geq k$ and positive natural numbers y_0, y_1 such that $m = y_0y + y_1(y+1)$ and $y_0 + y_1 = \ell z$. That is, ℓ denotes the number of iterations the topmost iterator does, and hence must be at least as big as k, while y_0 (respectively, y_1) denotes the number of times the inner iterator reads y (respectively, $y+1$) a's. We now show that such ℓ, y_0, and y_1 indeed exist.

Thereto, let ℓ be the biggest natural number such that $yz\ell \leq m$, and observe that, as $m \geq y^2zk$, it must hold that $\ell \geq y$ and $\ell \geq k$. Then, let $y_1 = m - yz\ell$ and $y_0 = \ell z - y_1$. It remains to verify that ℓ, y_0 and y_1 satisfy the desired conditions. We already observed that $\ell \geq k$, and, by definition of y_0, also $y_0 + y_1 = \ell z$. Further, to show that $m = y_0y + y_1(y+1)$, note that $m = yz\ell + y_1$ and thus $m = y(y_0 + y_1) + y_1 = y_0y + y_1(y+1)$. Finally, we must show that y_0 and y_1 are positive numbers. For y_1 this is clear as $y_1 = m - yz\ell$, and ℓ is chosen such that $yz\ell \leq m$. For y_0, recall that $\ell \geq y$, and, hence, $y_0 \geq yz - y_1$. Assuming y_0 to be negative implies that $yz \geq y_1 = m - yz\ell$. However, the latter implies that ℓ is not chosen to be maximal, as then also $yz(\ell+1) \leq m$, which is a contradiction.

This concludes the proof of Lemma 57. □

Theorem 59. Let Σ be an alphabet with $|\Sigma| \geq 2$. Then,

$$\mathrm{DRE}^{\#}_{S} \subsetneq \mathrm{DRE}^{\#}_{W}$$

Proof. By definition, $\mathrm{DRE}^{\#}_{S} \subset \mathrm{DRE}^{\#}_{W}$. Hence, it suffices to show that the inclusion is strict for a binary alphabet $\Sigma = \{a, b\}$. A witness for this strict inclusion is $r = (a^{2,3}(b + \varepsilon))^*$. As r is weakly deterministic, $L(r) \in \mathrm{DRE}^{\#}_{W}$. By applying the algorithm of Brüggemann-Klein and Wood on r for testing whether a regular language $L(r)$ is in DRE_W [17], it can be seen that $L(r) \notin \mathrm{DRE}_W$, and thus also not in DRE_S. By Theorem 54, we therefore have that $L(r) \notin \mathrm{DRE}^{\#}_S$. $\square$

Theorem 60. Let Σ be an arbitrary alphabet. Then,

$$\mathrm{DRE}^{\#}_W \subsetneq \mathrm{REG}(\Sigma)$$

Proof. Clearly, every language defined by a regular expression with counting is regular. Hence, it suffices to show that the inclusion is strict. We prove that the inclusion is strict already for a unary alphabet $\Sigma = \{a\}$. Thereto, consider the expression $r = (aaa)^*(a + aa)$. We can easily see that $L(r) \notin \mathrm{DRE}_W$ by applying the algorithm of Brüggemann-Klein and Wood [17] to $L(r)$. As r is over a unary alphabet, it follows from Theorems 54 and 55 that $L((aaa)^*(a + aa)) \notin \mathrm{DRE}^{\#}_W$, and hence $\mathrm{DRE}^{\#}_W \subsetneq \mathrm{REG}(\Sigma)$. $\square$

6.3 Succinctness

In Section 6.2 we learned that $\mathrm{DRE}^{\#}_W$ strictly contains $\mathrm{DRE}^{\#}_S$, prohibiting a translation from weakly to strongly 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 strongly deterministic one, constructs this expression. However, this is not the case:

Theorem 61. For every $n \in \mathbb{N}$, there exists an RE(#) expression r 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.

Before proving this theorem, we first give a lemma used in its proof.

Lemma 62. Let r be a strongly deterministic regular expression over alphabet $\{a\}$ with only one occurrence of a. Then $L(r)$ can be defined by one of the following expressions:

(1) $(a^{k,k})^{x,y}$

(2) $(a^{k,k})^{x,y} + \varepsilon$

where $k \in \mathbb{N}$, $x \in \mathbb{N} \cup \{0\}$, and $y \in \mathbb{N} \cup \{\infty\}$.

Proof. First, suppose that $L(r)$ does not contain ε. Since r has one occurrence of a, r is a nesting of iterators. However, since r is strongly deterministic, r can not have subexpressions of the form $s^{x,y}$ with $|L(s)| > 1$ and $(x, y) \neq (1, 1)$. It follows immediately that r is of the form $((\cdots(a^{k_1,k_1})^{k_2,k_2}\dots)^{k_n,k_n})^{x,y}$. Setting $k := k_1 \times \cdots \times k_n$ proves this case.

Second, suppose that $L(r)$ contains ε. If r is of the form $s_1 + s_2$, then one of s_1 and s_2, say s_2, does not contain an a, and hence $L(s_2) = \{\varepsilon\}$. Then, if $L(s)$ does not contain ε, we are done due to the above argument. So, the only remaining case is that r is a nesting of iterators, possibly defining ε. We can assume without loss of generality that r does not have subexpressions of the form $s^{1,1}$. Again, since r is strongly deterministic, it cannot have subexpressions of the form $s^{x,y}$ with $(x,y) \neq (0,1)$ and $|L(s) - \{\varepsilon\}| > 1$. Hence, by replacing subexpressions of the form $(s^{k,k})^{\ell,\ell}$ by $s^{k\ell,k\ell}$ and $(s^{0,1})^{0,1}$ by $s^{0,1}$, r can either be rewritten to $(a^{k,k})^{x,y}$ or $((a^{k,k})^{x,y})^{0,1}$. In the first case, the lemma is immediate and, in the second case, we can rewrite r as $((a^{k,k})^{x,y}) + \varepsilon$. $\square$

We are now ready to prove Theorem 61.

Proof. [of Theorem 61] Let r be the weakly deterministic expression $(a^{N+1,2N})^{1,2}$ for $N = 2^n$. Clearly, the size of r is $\mathcal{O}(n)$. Note that r defines all strings of length at least $N+1$ and at most $4N$, except a^{2N+1}. These expressions, in fact, where introduced by Kilpeläinen [63] when studying the inclusion problem for weakly deterministic expressions with counting.

We prove that every strongly deterministic expression for $L(r)$ is of size at least 2^n. Thereto, let s be a strongly deterministic regular expression for $L(r)$ with a minimal number of occurrences of the symbol a and, among those minimal expressions, one of minimal size.

We first argue that s is in a similar but more restricted normal form as the one we have in the proof of Lemma 57. If s is minimal and strongly deterministic, then

(a) if s has a disjunction of the form $s_1 + s_2$ then either $L(s_1) = \{\varepsilon\}$ or $L(s_2) = \{\varepsilon\}$;

(b) there are no subexpressions of the form $s_1 s_2$ in s in which $|L(s_1)| > 1$ or $\varepsilon \in L(s_1)$;

(c) there are no subexpressions of the form $s_1^{1,1}$, $(s_1^{0,1})^{0,1}$, $aa^{k,\ell}$, $a^{k,\ell}a$, $a^{k_1,\ell_1}a^{k_2,\ell_2}$, or $(a^{k,k})^{\ell,\ell}$;

(d) there are no subexpressions of the form $(s_1)^{k,\ell}$ with $(k,\ell) \neq (0,1)$ and $|L(s_1) - \{\varepsilon\}| > 1$;

The reasons are as follows:

- (a),(b): otherwise, s is not weakly deterministic;

- (c): otherwise, s is not minimal; and
- (d): otherwise, by (c), $\ell \geq 2$, which implies that s is not strongly deterministic.

Due to (a), we can assume without loss of generality that s does not have any disjunctions. Indeed, when $L(s_2) = \{\varepsilon\}$, we can replace every subexpression $s_1 + s_2$ or $s_2 + s_1$ with $(s_1)^{0,1}$ since the latter expression has the same or a smaller size as the former ones. From (a)–(d), and the fact that $\varepsilon \notin L(s)$, it then follows that s is of the form

$$s = s_1(s_2(\cdots(s_m)^{0,1}\cdots)^{0,1})^{0,1}$$

where each s_i ($i = 1, \ldots, m-1$) is of the form $a^{k,k}$. Notice that s_m can still be non-trivial according to (a)–(d), such as $(a^{7,7})^{8,9}$ or $a^{7,7}((a^{2,2})^{0,1})^{2,2}$. We now argue that s_m is either of the form s'_m or $s''_m s'_m$, where s'_m and s''_m have only one occurrence of the symbol a. We already observed above that s_m does not have any disjunctions. So, the only remaining operators are counting and conjunction.

Suppose, towards a contradiction, that s_m has at least two conjunctions (and, therefore, at least three occurrences of a). Because of (b) and (c), s_m cannot be of the form $p_1p_2p_3$, since then $|L(p_1p_2)| = 1$ and p_1p_2 can be rewritten. Hence, s_m is of the form p_1p_2, where either p_1 or p_2 is an iterator which contains a conjunction. If p_1 contains a conjunction we know due to (b) that $|L(p_1)| = 1$ and then p_1 can be rewritten. If p_2 is an iterator that contains a conjunction then p_2 is of the form $p_3^{0,1}$ due to (d) and since $|L(s_m)| = \infty$. Due to (c), p_3 cannot be of the form $p_4^{0,1}$. Due to (d), p_3 cannot be of the form $p_4^{k,\ell}$ with $(k,\ell) \neq (0,1)$. Hence, $p_3 = p_4p_5$. But this means that $s_m = p_1(p_4p_5)^{0,1}$ which violates the definition of s_m, being the innermost expression in the normal form for s above (i.e., s_m should have been p_4p_5). This shows that s_m has at most one conjunction.

It now follows analogously as in the reasoning for p_3 above that s_m is either of the form s'_m or $s''_m s'_m$, where s'_m and s''_m have only one occurrence of the symbol a.

We will now argue that, for each string of the form $a^{N+1}, \ldots, a^{2N}$, its last position is matched onto a different symbol in s. Formally, let $\overline{u}_{N+1}, \ldots, \overline{u}_{2N}$ be the unique strings in $L(\overline{s})$ such that $|\overline{u}_i| = i$, for every $i \in [N+1, 2N]$. We claim that the last symbol in each $\overline{u}_i$ is different. This implies that s contains at least N occurrences of a, making the size of s at least $N = 2^n$, as desired. Thereto, let $\overline{w}_1$ and $\overline{w}_2$ be two different strings in $\{\overline{u}_{N+1}, \ldots, \overline{u}_{2N}\}$. Without loss of generality, assume $\overline{w}_1$ to be the shorter string. Towards a contradiction, assume that $\overline{w}_1$ and $\overline{w}_2$ end with the same symbol $\overline{x}$. Let

$\overline{s} = \overline{s}_1(\overline{s}_2(\cdots(\overline{s}_m)^{0,1}\cdots)^{0,1})^{0,1}$. Due to the structure of $\overline{s}$ and since both $\overline{w}_1$ and $\overline{w}_2$ are in $L(\overline{s})$, this implies that $\overline{x}$ cannot occur in $\overline{s}_1, \ldots, \overline{s}_{m-1}$ and that $\overline{x}$ must be the rightmost symbol in $\overline{s}$. As $\overline{s}_m$ is either of the form $\overline{s}'_m$ or $\overline{s}''_m\overline{s}'_m$ this implies $\overline{x}$ occurs in $\overline{s}'_m$. Again due to the structure of s and s_m, this means that s'_m must always define a language of the form $\{w \mid vw \in L(r)\}$, where v is a prefix of $\mathrm{dm}(\overline{w}_1)$. Considering the language defined by s, $L(s'_m)$ hence contains the strings $a^i, \ldots, a^{i+k}, a^{i+k+2}, \ldots, a^{i+k+2N}$ for some $i \geq 1$ and $k \geq 0$. However, as s'_m only contains a single occurrence of the symbol a, Lemma 62 says that it cannot define such a language. This is a contradiction.

It follows that the size of s is at least 2^n. □

6.4 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 Guard(C), as follows: for every cv $\in C$ and $k \in \mathbb{N}$, we have that true, false, cv $= k$, and cv $< k$ are in Guard(C). Moreover, when $\phi_1, \phi_2 \in$ Guard(C), then so are $\phi_1 \wedge \phi_2$, $\phi_1 \vee \phi_2$, and $\neg\phi_1$. For $\phi \in$ 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 cv $\in C$ occurs at most once. By Update(C) we denote the set of all updates.

Definition 63. 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; $\delta : Q \times \Sigma \times \mathrm{Guard}(C) \times \mathrm{Update}(C) \times Q$ is the transition relation; $F : Q \to \mathrm{Guard}(C)$ 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 cv $:= 1$, when reset(cv) $\in \pi$, and cv $:=$ cv $+ 1$ when cv++ $\in \pi$ and $\alpha(\mathrm{cv}) < \tau(\mathrm{cv})$. 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 $\gamma \to_a \gamma'$, 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 $\gamma \to_{a_1} \cdots \to_{a_n} \gamma'$. 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 $\alpha \models \phi$. Note that, as this definition only concerns reachable configurations, it is not straightforward to test whether an CNFA is deterministic (as will also become clear in Theorem 64(6)). However, as non-reachable configurations have no influence on the determinism of runs on the automaton, we choose not to take them into account.

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 64.
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.

Proof. We first note that a CNFA $A = (Q, q_0, C, \delta, F, \tau)$ can easily be completed. For $q \in Q$, $a \in \Sigma$ define $\text{Formulas}(q, a) = \{\phi \mid (q, a, \phi, \pi, q') \in \delta\}$. We say that A is *complete* if, for any value function α, it holds that $\alpha \models \bigvee_{\phi \in \text{Formulas}(q,a)} \phi$. That is, for any configuration (q, α) and symbol a, there is always a transition which can be followed by reading a.

Define the completion A^c of A as $A^c = (Q^c, q_0, C, \delta^c, F^c, \tau^c)$ with $Q^c = Q \cup \{q_s\}$, δ^c is δ extended with $(q_s, a, \text{true}, \emptyset, q_s)$ and, for all $q \in Q$, $a \in \Sigma$, with the tuples $(q, a, \phi^c_{a,q}, \phi, q_s)$, where $\phi^c_{a,q} = \neg \bigvee_{\phi \in \text{Formulas}(q,a)} \phi$. Finally, F^c is F extended with $F^c(q_s) = \text{false}$. Note also that A^c is deterministic if and only if A is deterministic. Hence, from now on we can assume that all CNFAs and CDFAs under consideration are complete. We now prove the six statements.

(1) Given two complete CNFAs A_1, A_2, where $A_i = (Q_i, q_i, C_i, \delta_i, F_i, \tau_i)$, their union can be defined as $A = (Q_1 \times Q_2, (q_1, q_2), C_1 \uplus C_2, \delta, F, \tau_1 \cup \tau_2)$. Here,

- $\delta = \{((s_1, s_2), a, \phi_1 \wedge \phi_2, \pi_1 \cup \pi_2, (s'_1, s'_2)) \mid (s_i, a, \phi_i, \pi_i, s'_i) \in \delta_i \text{ for } i = 1, 2\}$; and
- $F(s_1, s_2) = F_1(s_1) \vee F_2(s_2)$.

For the intersection of A_1 and A_2, the definition is completely analogue, only the acceptance condition differs: $F(s_1, s_2) = F(s_1) \wedge F(s_2)$. It is easily verified that if A_1 and A_2 are both deterministic, then A is also deterministic.

(2) Given a complete CDFA $A = (Q, q, C, \delta, F, \tau)$, the CDFA $A' = (Q, q, C, \delta, F', \tau)$, where $F'(q) = \neg F(q)$, for all $q \in Q$, defines the complement of A.

(3) Since A is a CDFA, from every reachable configuration only one transition can be followed when reading a symbol. Hence, we can match the string w from left to right, maintaining at any time the current configuration of A. Hence, we need to make $|w| + 1$ transitions, while every transition can be made in time $\mathcal{O}(|A|)$. Therefore, testing MEMBERSHIP can be done in time $\mathcal{O}(|w||A|)$.

(4) Obviously, we can decide in non-deterministic polynomial time whether a string w is accepted by a CNFA A. It suffices to guess a sequence of $|w|$ transitions, and check whether this sequence forms a valid run of A on w.

To show that it is NP-hard, we do a reduction from BIN PACKING [36]. This is the problem, given a set of weights $W = \{n_1, \ldots, n_k\}$, a packet size p, and the maximum number of packets m; decide whether there is a partitioning of $W = S_1 \uplus S_2 \uplus \cdots \uplus S_m$ such that for each $i \in [1, m]$, we have that $\sum_{n \in S_i} n \leq p$. The latter problem is NP-complete, even when all integers are given in unary.

We will construct a string w and a CNFA A such that $w \in L(A)$ if and only if there exists a proper partitioning for $W = \{n_1, \ldots, n_k\}$, p, and m. Let $\Sigma = \{\#, 1\}$, and $w = \#1^{n_1}\#\#1^{n_2}\# \cdots \#1^{n_k}\#$. Further, A will have an initial state q_0 and for every $j \in [1, m]$ a state q_j and countervariable cv_j. When reading the string w the automaton will non-deterministically guess for each n_i to which of the m sets n_i is assigned (by going to its corresponding state) and maintaining the running sum of the different sets in the countervariables. (As the initial value of the countervariables is 1, we actually store the running sum plus 1) In the end it can then easily be verified whether the chosen partitioning satisfies the desired properties.

Formally, $A = (Q, q_0, \{\mathrm{cv}_1, \ldots, \mathrm{cv}_m\}, \delta, F, \tau)$ can be defined as follows:

- $Q = \{q_0, \ldots, q_m\}$;
- For all $i \in [1, m]$, $(q_0, \#, \text{true}, \emptyset, q_i), (q_i, 1, \text{true}, \{\mathrm{cv}_i{+}{+}\}, q_i), (q_i, \#, \text{true}, \emptyset, q_0) \in \delta$;
- $F(q_0) = \bigwedge_{i \in [1,m]} \mathrm{cv}_i \leq p + 1$, and for $q \neq q_0$, $F(q) = \text{false}$; and
- for all $i \in [1, m]$, $\tau(\mathrm{cv}_i) = p + 2$.

(5) We show that EMPTINESS is in PSPACE for CNFAs, and PSPACE-hard for CDFAs. The theorem then follows.

The algorithm for the upperbound guesses a string w which is accepted by A. Instead of guessing w at once, we guess it symbol by symbol and store one configuration γ, such that A can be in γ after reading the already guessed string w. When γ is an accepting configuration, we have guessed a string which is accepted by A, and have thus shown that $L(A)$ is non-empty. Since any configuration can be stored in space polynomial in A (due to the maximum values τ on the countervariables), we hence have a non-deterministic polynomial space algorithm for the complement of the EMPTINESS problem. As NPSPACE = PSPACE, and PSPACE is closed under complement, it follows that EMPTINESS for CNFAs is in PSPACE.

We show that the EMPTINESS problem for CDFAs is PSPACE-hard by a reduction from REACHABILITY of 1-safe Petri nets.

A *net* is a triple $N = (S, T, E)$, where S and T are finite sets of *places* and *transitions*, and $E \subseteq (S \times T) \cup (T \times S) \to \{0, 1\}$. The *preset* of a transition t is denoted by $\bullet t$, and defined by $\bullet t = \{s \mid E(s, t) = 1\}$. A *marking* is a mapping $M : S \to \mathbb{N}$. A *Petri net* is a pair $\mathcal{N} = (N, M_0)$, where N is a net and M_0 is the initial marking. A transition t is enabled at a marking M if $M(s) > 0$ for every $s \in \bullet t$. If t is enabled at M, then it can *fire*, and its firing leads to the successor marking M' which is defined for every place s by $M'(s) = M(s) + E(t, s) - E(s, t)$. The expression $M \to_t M'$ denotes that M enables transition t, and that the marking reached by the firing of t is M'. Given a (firing) sequence $\sigma = t_1 \cdots t_n$, $M \Rightarrow_\sigma M'$ denotes that there exist markings $M_1, M_2, \ldots, M_{n-1}$ such that $M \to_{t_1} M_1 \cdots M_{n-1} \to_{t_n} M'$. A Petri net is *1-safe* if $M(s) \leq 1$ for every place s and every reachable marking M.

REACHABILITY for 1-safe Petri nets is the problem, given a 1-safe Petri net (N, M_0) and a marking M, is M reachable from M_0. That is, does there exist a sequence σ such that $M_0 \Rightarrow_\sigma M$. The latter problem is PSPACE-complete [32].

We construct a CDFA A such that $L(A) = \emptyset$ if and only if M is not reachable from M_0. That is, A will accept all strings $at_1 \cdots t_k$ such that

$M_0 \Rightarrow_{t_1 \cdots t_k} M$. To achieve this, A will simulate the working of the Petri net on the firing sequence given by the string. Therefore, we maintain a countervariable for every place s, which will have value 1 when $M'(s) = 0$ and 2 when $M'(s) = 1$, where M' is the current marking. With every transition of the Petri net, we associate a transition in A. Here, the guard ϕ is used to check whether the preconditions for the firing of this transition are satisfied, and the updates are used to update the values of the countervariables according to the transition. The string is accepted if after executing this firing sequence, the countervariables correspond to the given marking M.

Formally, let $N = (S, T, E)$. Then, the CDFA $A = (\{q_0, q_1\}, q_0, S, \delta, F, \tau)$ is defined as follows:

- $(q_0, a, \text{true}, \pi, q_1) \in \delta$, where $\pi = \{s\text{++} \mid M_0(s) = 1\}$;
- For every $t \in T$: $(q_1, t, \phi, \pi, q_1) \in \delta$, where $\phi = \bigwedge_{s \in \bullet t} s = 2$ and $\pi = \{\text{reset}(s) \mid E(s,t) > E(t,s)\} \cup \{s\text{++} \mid E(t,s) > E(s,t)\}$;
- $F(q_0) = \text{false}$, and $F(q_1) = \bigwedge_{M(s)=0} s = 1 \wedge \bigwedge_{M(s)=1} s = 2$; and
- For every $s \in S$, $\tau(s) = 2$.

(6) To show that it is in PSPACE, we guess a string w character by character and only store the current configuration. If at any time it is possible to follow more than one transition at once, A is non-deterministic. Again, since NPSPACE = PSPACE and PSPACE is closed under complement, the result follows.

To show that the problem is PSPACE-hard, we do a reduction from ONE RUN of 1-safe Petri nets. This is the problem, given a 1-safe Petri net $\mathcal{N} = (N, M_0)$, is there exactly one run on N. The latter problem is PSPACE-complete [32].

We construct a CNFA A which is deterministic if and only if ONE RUN is true for $\mathcal{N}$. To do this, we set the alphabet of A to $\{a, t\}$ and A will accept strings of the form at^*, and simulates the working of $\mathcal{N}$. The set of states $Q = \{q_0, q_c, t_1, \ldots, t_n\}$, when $T = \{t_1, \ldots, t_n\}$. Furthermore, there is one countervariable for every place: $C = S$.

The automaton now works as follows. From its initial state q_0 it goes to its central state q_c and sets the values of the countervariables to the initial marking M_0. From q_c, there is a transition to every state t_i. This transition can be followed by reading a t if and only if the values of the countervariables satisfy the necessary conditions to fire the transition t_i. Then, from the states t_i, there is a transition back to q_c, which can be followed by reading a t and is used to update the countervariables according to the firing of t_i. As in the previous proof, the countervariable for place s will have value 1 when $M'(s) = 0$ and 2 when $M'(s) = 1$, where M' is the current marking.

More formally, $A = (\{q_0, q_c\} \cup T, q_0, S, \delta, F, \tau)$ is constructed as follows:

- $(q_0, a, \text{true}, \pi, q_c) \in \delta$, where $\pi = \{s{+}{+} \mid M_0(s) = 1\}$;
- For all $t_i \in T$, $(q_c, t, \phi, \emptyset, t_i) \in \delta$, where $\phi = \bigwedge_{s \in \bullet t_i} s = 2$;
- For all $t_i \in T$, $(t_i, t, \text{true}, \pi, q_c) \in \delta$, where $\pi = \{\text{reset}(s) \mid E(s, t_i) > E(t_i, s)\} \cup \{s{+}{+} \mid E(t_i, s) > E(s, t_i)\}$;
- For all $s \in S$, $\tau(s) = 2$; and
- $F(q) = \text{true}$, for all $q \in Q$.

□

6.5 From RE(#) to CNFA

In this section, we show how an RE(#) 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 [15]. We emphasize at this point that such an extended Glushkov construction has already been given by Sperberg-McQueen [101]. 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(#) expressions. We refer to G_r as the *Glushkov counting automaton of* r.

We first provide some notation and terminology needed in the construction below. For a marked expression $\overline{r}$, a marked symbol $\overline{x}$ and an iterator c of $\overline{r}$ we denote by $\text{iterators}_{\overline{r}}(\overline{x}, c)$ the list of all iterators of c which contain $\overline{x}$, except c itself. For marked symbols $\overline{x}, \overline{y}$, we denote by $\text{iterators}_{\overline{r}}(\overline{x}, \overline{y})$ all iterators of $\overline{r}$ which contain $\overline{x}$ but not $\overline{y}$. Finally, let $\text{iterators}_{\overline{r}}(\overline{x})$ be the list of all iterators of $\overline{r}$ which contain $\overline{x}$. Note that all such lists $[c_1, \ldots, c_n]$ contain a sequence of nested subexpressions. Therefore, we will always assume that they are ordered such that $c_1 \prec c_2 \prec \cdots \prec c_n$. That is, they form a sequence of nested subexpressions. Moreover, whenever $\overline{r}$ is clear from the context, we omit it as a subscript. 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 $\text{iterators}(a_1) = [a_1^{1,2}, (a_1^{1,2} b_1)^{3,4}, ((a_1^{1,2} b_1)^{3,4})^{5,6}]$.

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 $c \neq$ null, this transition iterates over c and all iterators in iterators$(\overline{x}, c)$ are reset by going to $\overline{y}$. Otherwise, if c equals null, the iterators in iterators$(\overline{x}, \overline{y})$ are reset. Formally, the set follow$(\overline{r})$ contains for each subexpression $\overline{s}$ of $\overline{r}$,

- all tuples $(\overline{x}, \overline{y}, \text{null})$ for $\overline{x}$ in last$(\overline{s}_1)$, $\overline{y}$ in 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 last$(\overline{s}_1)$, $\overline{y}$ in first$(\overline{s}_1)$, and $\overline{s} = \overline{s}_1^{k,\ell}$.

We introduce a counter variable 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:

- value-test$([c_1, \ldots, c_n]) := \bigwedge_{c_i}(\text{lower}(c_i) \leq \text{cv}(c_i)) \wedge (\text{cv}(c_i) \leq \text{upper}(c_i))$. When we leave the iterators $c_1, \ldots, c_n$ we have to check that we have done an admissible number of iterations for each iterator.
- upperbound-test(c) := cv$(c) <$ upper(c) when c is a bounded iterator and, otherwise, upperbound-test(c) := true. When iterating over a bounded iterator, we have to check that we can still do an extra iteration.
- reset$([c_1, \ldots, c_n]) := \{\text{reset}(\text{cv}(c_1)), \ldots, \text{reset}(\text{cv}(c_n))\}$. 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 $C = \{\text{cv}(c) \mid c \text{ is an iterator of } \overline{r}\}$. We next define the transition function. For all $\overline{y} \in$ first$(\overline{r})$, $(q_0, \text{dm}(\overline{y}), \mathit{true}, \emptyset, q_{\overline{y}}) \in \delta$.[4] For every element $(\overline{x}, \overline{y}, c) \in$ follow$(\overline{r})$, we define a transition $(q_{\overline{x}}, \text{dm}(\overline{y}), \phi, \pi, q_{\overline{y}}) \in \delta$. If $c =$ null, then $\phi :=$ value-test(iterators$(\overline{x}, \overline{y})$) and $\pi :=$ reset(iterators$(\overline{x}, \overline{y})$). If $c \neq$ null, then $\phi :=$ value-test(iterators$(\overline{x}, c)$) $\wedge$ upperbound-test(c) and $\pi :=$ reset(iterators$(\overline{x}, c)$) $\cup$ update(c). The acceptance criteria of G_r depend on the set last$(\overline{r})$. For any symbol $\overline{x} \notin$ last$(\overline{r})$, $F(q_{\overline{x}})$:= false. For every element $\overline{x} \in$ last$(\overline{r})$, $F(q_{\overline{x}})$:= value-test(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, $F(q_0)$:= true if $\varepsilon \in L(r)$. Lastly, for all bounded iterators

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

c, $\tau(\mathrm{cv}(c)) = \mathrm{upper}(c)$ since c never becomes larger than upper(c), and for all unbounded iterators c, $\tau(\mathrm{cv}(c)) = \mathrm{lower}(c)$ as there are no upper bound tests for cv(c).

Theorem 65. For every RE(#) expression r, $L(G_r) = L(r)$. Moreover, G_r is deterministic if and only if r is strongly deterministic.

This theorem will largely follow from Lemma 66, for which we first introduce some notation. The goal will be to associate transition sequences of G_r with correctly bracketed words in $\widetilde{\overline{r}}$, the bracketing of $\overline{r}$. A *transition sequence* $\sigma = t_1, \ldots, t_n$ of G_r is simply a sequence of transitions of G_r. It is *accepting* if there exists a sequence of configurations $\gamma_0\gamma_1, \ldots, \gamma_n$ such that γ_0 is the inital configuration, γ_n is a final configuration, and, for every $i = 1, \ldots, n-1$, $\gamma_{i-1} \rightarrow_{a_i} \gamma_i$ by using transition $t_i = (q_{i-1}, a_i, \phi_i, \pi_i, q_i)$. Here, for each i, $\gamma_i = (q_i, \alpha_i)$.

Recall that the bracketing $\widetilde{\overline{r}}$ of an expression $\overline{r}$ is obtained by associating indices to iterators in $\overline{r}$ and by replacing each iterator $c := \overline{s}^{k,\ell}$ of $\overline{r}$ with index i with $([_i s]_i)^{k,\ell}$. In the following, we use ind(c) to denote the index i associated to iterator c. As every triple $(\overline{x}, \overline{y}, c)$ in follow($\overline{r}$) corresponds to exactly one transition t in G_r, we also say that t *is generated by* $(\overline{x}, \overline{y}, c)$.

We now want to translate transition sequences into correctly bracketed words. Thereto, we first associate a bracketed word $\mathrm{word}_{G_r}(t)$ to each transition t of G_r. We distinguish a few cases:

(i) If $t = (q_0, a, \phi, \pi, q_{\overline{y}})$, with iterators$(\overline{y}) = [c_1, \ldots, c_n]$, then $\mathrm{word}_{G_r}(t) = [_{\mathrm{ind}(c_n)} \cdots [_{\mathrm{ind}(c_1)} \overline{y}$.

(ii) If $t = (q_{\overline{x}}, a, \phi, \pi, q_{\overline{y}})$ and is generated by $(\overline{x}, \overline{y}, c) \in$ follow($\overline{r}$), with $c \neq$ null. Let iterators$(\overline{x}, c) = [c_1, \ldots, c_n]$ and iterators$(\overline{y}, c) = [d_1, \ldots, d_m]$. Then, $\mathrm{word}_{G_r}(t) =]_{\mathrm{ind}(c_1)} \cdots]_{\mathrm{ind}(c_n)} [_{\mathrm{ind}(d_m)} \cdots [_{\mathrm{ind}(d_1)} \overline{y}$.

(iii) If $t = (q_{\overline{x}}, a, \phi, \pi, q_{\overline{y}})$ and is generated by $(\overline{x}, \overline{y}, \mathrm{null}) \in$ follow($\overline{r}$). Let iterators$(\overline{x}, \overline{y}) = [c_1, \ldots, c_n]$ and let iterators$(\overline{y}, \overline{x}) = [d_1, \ldots, d_m]$. Then, $\mathrm{word}_{G_r}(t) =]_{\mathrm{ind}(c_1)} \cdots]_{\mathrm{ind}(c_n)} [_{\mathrm{ind}(d_m)} \cdots [_{\mathrm{ind}(d_1)} \overline{y}$.

Finally, to a marked symbol $\overline{x}$ with iterators$(\overline{x}) = [c_1, \ldots, c_n]$, we associate $\mathrm{word}_{G_r}(\overline{x}) =]_{\mathrm{ind}(c_1)} \cdots]_{\mathrm{ind}(c_n)}$. Now, for a transition sequence $\sigma = t_1, \ldots, t_n$, where $t_n = (q_{\overline{x}}, a, \phi, \pi, q_{\overline{y}})$, we set $\text{brack-seq}_{G_r}(\sigma) = \mathrm{word}_{G_r}(t_1) \cdots \mathrm{word}_{G_r}(t_n)$ $\mathrm{word}_{G_r}(\overline{y})$. Notice that $\text{brack-seq}_{G_r}(\sigma)$ is a bracketed word over a marked alphabet. We sometimes also say that σ *encodes* $\text{brack-seq}_{G_r}(\sigma)$. We usually omit the subscript G_r from word and brack-seq when it is clear from the context.

For a bracketed word $\widetilde{w}$, let strip(w) denote the word obtained from $\widetilde{w}$ by removing all brackets. Then, for a transition sequence σ, define run(σ) = strip(brack-seq(σ)).

Lemma 66. Let $\overline{r}$ be a marked regular expression with counting and G_r the corresponding counting Glushkov automaton for r.

(1) For every string $\widetilde{\overline{w}}$, we have that $\widetilde{\overline{w}}$ is a correctly bracketed word in $L(\widetilde{\overline{r}})$ if and only if there exists an accepting transition sequence σ of G_r such that $\widetilde{\overline{w}} = \text{brack-seq}(\sigma)$.

(2) $L(\overline{r}) = \{\text{run}(\sigma) \mid \sigma \text{ is an accepting transition sequence of } G_r\}$.

Proof. We first prove (1) by induction on the structure of $\overline{r}$. First, notice that brack-seq(σ) is a correctly bracketed word for every accepting transition sequence σ on G_r. Hence, we can restrict attention to correctly bracketed words below.

For the induction below, we first fix a bracketing $\widetilde{\overline{r}}$ of r and we assume that all expressions $\widetilde{\overline{s}}$, $\widetilde{\overline{s}}_1$, $\widetilde{\overline{s}}_2$ are correctly bracketed subexpressions of $\widetilde{\overline{r}}$.

- $\overline{s} = \overline{x}$. Then, also $\widetilde{\overline{s}} = \overline{x}$, and $L(\widetilde{\overline{s}}) = \overline{x}$. It is easily seen that the only accepting transition sequence σ of G_s consists of one transition t, with brack-seq$(\sigma) = \overline{x}$.

- $\overline{s} = \overline{s}_1 + \overline{s}_2$. Clearly, the set of all correctly bracketed words in $L(\widetilde{\overline{s}})$ is the union of all correctly bracketed words in $L(\widetilde{\overline{s}}_1)$ and $L(\widetilde{\overline{s}}_2)$. Further, observe that G_s is constructed from G_{s_1} and G_{s_2} by identifying their initial states and taking the disjoint union otherwise. As the initial states only have outgoing, and no incoming, transitions, the set of accepting transition sequences of G_s is exactly the union of the accepting transition sequences of G_{s_1} and G_{s_2}. Hence, the lemma follows from the induction hypothesis.

- $\overline{s} = \overline{s}_1 \cdot \overline{s}_2$. Let $\widetilde{\overline{w}}$ be a correctly bracketed word. We distinguish a few cases. First, assume $\widetilde{\overline{w}} \in \text{Char}(\widetilde{\overline{s}}_1)^*$, i.e., $\widetilde{\overline{w}}$ contains only symbols of $\widetilde{\overline{s}}_1$. Then, if $\varepsilon \notin L(s_2)$, $\widetilde{\overline{w}} \notin L(\widetilde{\overline{s}})$, and, by construction of G_s, there is no accepting transition sequence σ on G_s such that brack-seq$(\sigma) = \widetilde{\overline{w}}$. This is due to the fact that, with $\varepsilon \notin L(\overline{s}_2)$, for all $\overline{x} \in \text{Char}(\overline{s}_2)$, we have that $\overline{x} \notin \text{last}(\overline{s})$ and hence $F(q_{\overline{x}})$ = false. Hence, assume $\varepsilon \in L(\overline{s}_2)$. Then, $\widetilde{\overline{w}} \in L(\widetilde{\overline{s}})$ if and only if $\widetilde{\overline{w}} \in L(\widetilde{\overline{s}}_1)$ if and only if, by the induction hypothesis, there is an accepting transition sequence σ on G_{s_1}, with brack-seq$(\sigma) = \widetilde{\overline{w}}$. By construction of G_s, and the fact that $\varepsilon \in L(\overline{s}_2)$, brack-seq$(\sigma) = \widetilde{\overline{w}}$ if and only if σ is also an accepting transition sequence on G_s. This settles the case $\widetilde{\overline{w}} \in \text{Char}(\widetilde{\overline{s}}_1)^*$. The case $\widetilde{\overline{w}} \in \text{Char}(\widetilde{\overline{s}}_2)^*$ can be handled analogously.

Finally, consider the case that $\widetilde{\overline{w}}$ contains symbols from both $\widetilde{\overline{s}}_1$ and $\widetilde{\overline{s}}_2$. If $\widetilde{\overline{w}}$ is not of the form $\widetilde{\overline{w}} = \widetilde{\overline{w}}_1\widetilde{\overline{w}}_2$, with $w_1 \in \text{Char}(\widetilde{\overline{s}}_1)^*$ and $w_2 \in \text{Char}(\widetilde{\overline{s}}_2)^*$, then we immediately have that $\widetilde{\overline{w}} \notin L(\widetilde{\overline{s}})$ nor can there be a transition sequence σ on G_s encoding $\widetilde{\overline{w}}$. Hence, assume $\widetilde{\overline{w}}$ is of this form. Then, $\widetilde{\overline{w}} \in L(\widetilde{\overline{s}})$ if and only if $\widetilde{\overline{w}}_i \in L(\widetilde{\overline{s}}_i)$, for $i = 1, 2$ if and only if, by the induction hypothesis, there exist accepting transition sequences σ_1 (resp. σ_2) on G_{s_1} (resp. G_{s_2}) encoding $\widetilde{\overline{w}}_1$ (resp. $\widetilde{\overline{w}}_2$). Let $\sigma_1 = t_1, \ldots, t_n$ and $\sigma_2 = t_{n+1}, \ldots, t_m$, with $q_{\overline{x}}$ the target state of t_n, and $q_{\overline{y}}$ the target state of t_{n+1}. Further, let $t = (q_{\overline{x}}, a, \phi, \pi, q_{\overline{y}})$ be the unique transition of G_s generated by the tuple $(\overline{x}, \overline{y}, \text{null}) \in \text{follow}(\overline{s})$. We claim that $\sigma = t_1, \ldots, t_n, t, t_{n+2}, \ldots, t_m$ is an accepting transition sequence on G_s with brack-seq(σ) = brack-seq(σ_1)brack-seq(σ_2), and hence brack-seq$(\sigma) = \widetilde{\overline{w}}$. To see that σ is an accepting transition sequence on G_s note that the guard $F(q_{\overline{x}})$ (in G_{s_1}) is equal to ϕ, the guard of t. Hence, the fact that σ_1 is an accepting transition sequence on G_{s_1} ensures that t can be followed in G_s. Further, note that after following transition t, all counters are reset, as they were after following t_{n+1} in G_{s_2}. This ensures that σ_1 and σ_2 can indeed be composed to the accepting transition sequence σ in this manner. Further, to see that brack-seq$_{G_s}(\sigma)$ = brack-seq$_{G_{s_1}}(\sigma_1)$brack-seq$_{G_{s_2}}(\sigma_2)$ it suffices to observe that word$_{G_{s_1}}(q_{\overline{x}})$word$_{G_{s_2}}(t_{n+1})$ = word$_{G_s}(t)$, by definition. Hence, σ is an accepting transition sequence on G_s, with brack-seq$_{G_s}(\sigma) = \widetilde{\overline{w}}$, as desired. Conversely, we need to show that any such transition sequence σ can be decomposed in accepting transition sequences σ_1 and σ_2 satisfying the same conditions. This can be done using the same reasoning as above.

- $\overline{s} = \overline{s}_1^{[k,\ell]}$. Let $\widetilde{\overline{s}} = ([_j\widetilde{\overline{s}}_1]_j)^{k,\ell}$ and $\widetilde{\overline{w}}$ be a correctly bracketed word.

First, assume $\widetilde{\overline{w}} \in L(\widetilde{\overline{s}})$, we show that there is an accepting transition sequence σ on G_s encoding $\widetilde{\overline{w}}$. As $\widetilde{\overline{w}}$ is correctly bracketed, we can write $\widetilde{\overline{w}} = [_j\widetilde{\overline{w}}_1]_j[_j\widetilde{\overline{w}}_2]_j \cdots [_j\widetilde{\overline{w}}_n]_j$, with $n \in [k, \ell]$, and $\widetilde{\overline{w}}_i \in L(\widetilde{\overline{s}}_1)$, $\widetilde{\overline{w}}_i \neq \varepsilon$, for all $i \in [1, n]$. Then, by the induction hypothesis, there exist valid transition sequences $\sigma_1, \ldots, \sigma_n$ on G_{s_1} encoding w_i, for each $i \in [1, n]$. For all $i \in [1, n]$, let $\sigma_i = t_1^i, \ldots, t_{m_i}^i$, for some m_i, the target state of t_1^i be $q_{\overline{y}_i}$ and the target state of $t_{m_i}^i$ be $q_{\overline{x}_i}$. For all $i \in [1, n-1]$, let t^i be the unique transition generated by $(\overline{x}_i, \overline{y}_{i+1}, \overline{s}) \in \text{follow}(\overline{s})$, and define $\sigma = t_1^1, \ldots, t_{m_1}^1, t^1, t_2^2, \ldots, t_{m_2}^2, t^2, t_2^3, \ldots, t^{n-1}, t_2^n, \ldots, t_{m_n}^n$. It now suffices to show that σ is an accepting transition sequence on G_s and brack-seq(σ) $= \widetilde{\widetilde{w}}$. The reasons are analogous to the ones for the constructed transition sequence σ in the previous case ($\overline{s} = \overline{s}_1 \cdot \overline{s}_2$). To see that σ is an accepting transition sequence, note that we simply execute the different transition sequences σ_1 to σ_n one after another, separated by iterations over the topmost iterator $\overline{s}$, by means of the transitions t^1 to t^n. As these transitions reset all counter variables (except cv($\overline{s}$)) the counter variables on each of these separate runs always have the same values as they had in the runs on G_{s_1}. Therefore, it suffices to argue that the transitions t^i can safely be followed in the run σ, and that we finally arrive in an accepting configuration. This is both due to the fact that we do n such iterations, with $n \in [k, \ell]$, and each iteration increments cv($\overline{s}$) by exactly 1. To see that brack-seq$_{G_s}(\sigma) = \widetilde{\widetilde{w}}$, note that, by induction, $\widetilde{\widetilde{w}} =$ $[_j$brack-seq$_{G_{s_1}}(\sigma_1)]_j[_j$brack-seq$_{G_{s_1}}(\sigma_2)]_j \cdots [_j$brack-seq$_{G_{s_1}}(\sigma_n)]_j$. Therefore, it suffices to observe that word$_{G_s}(t_1^1) = [_j$word$_{G_{s_1}}(t_1^1)$, word$_{G_s}(t_{m_n}^n)$ $=$ word$_{G_{s_1}}(t_{m_n}^n)]_j$, for all $i \in [1, n-1]$, word$_{G_s}(t^i)$ $=$ word$_{G_{s_1}}(\overline{x}_i)]_j[_j$ word$_{G_{s_1}}(t_1^{i+1})$, and word$_{G_s}(t)$ $=$ word$_{G_{s_1}}(t)$, for all other transitions t occurring in σ.

Conversely, assume that there exists an accepting transition sequence σ on G_s encoding $\widetilde{\widetilde{w}}$. We must show $\widetilde{\widetilde{w}} \in L(\widetilde{\overline{s}})$. This can be done using arguments analogous to the ones above. It suffices to note that σ contains n transitions t^1 to t^n, for some $n \in [k, \ell]$, generated by a tuple of the form $(\overline{x}, \overline{y}, \overline{s}) \in$ follow($\overline{s}$). This allows to decompose σ into n accepting transition sequences σ_1 to σ_n and apply the induction hypothesis to obtain the desired result.

To prove the second point, i.e. $L(\overline{s}) = \{$run$(\sigma) \mid \sigma$ is an accepting transition sequence on $G_s\}$, it suffices to observe that

$$\begin{aligned} L(\overline{s}) &= \{\text{strip}(\widetilde{\widetilde{w}}) \mid \widetilde{\widetilde{w}} \in L(\widetilde{\overline{s}}) \text{ and } \widetilde{\widetilde{w}} \text{ correctly bracketed}\} \\ &= \{\text{strip}(\text{brack-seq}(\sigma)) \mid \sigma \text{ is an accepting transition sequence on } G_s\} \\ &= \{\text{run}(\sigma) \mid \sigma \text{ is an accepting transition sequence on } G_s\} \end{aligned}$$

Here, the first equality follows immediately from Lemma 67 below, the second equality from the first point of this lemma, and the third is immediate from the definitions. □

The following lemma, which we state without proof, is immediate from the definitions. However, notice that it is important in this lemma that, for subexpressions $s^{k,\ell}$ with s nullable, we have $k = 0$, as required by the normal form for RE(#).

Lemma 67. Let $r \in \mathrm{RE}(\#)$, and $\widetilde{r}$ be the bracketing of r. Then,

$$L(r) = \{\mathrm{strip}(\widetilde{w}) \mid \widetilde{w} \in L(\widetilde{r}) \text{ and } \widetilde{w} \text{ is correctly bracketed}\}.$$

We are now ready to prove Theorem 65.

Proof. [of Theorem 65] Let $r \in \mathrm{RE}(\#)$, $\overline{r}$ a marking of r, and G_r its corresponding counting Glushkov automaton.

We first show that $L(r) = L(G_r)$. Thereto, observe that (1) $L(r) = \{\mathrm{dm}(\overline{w}) \mid \overline{w} \in L(\overline{r})\}$, by definition of $\overline{r}$, and (2) $L(G_r) = \{\mathrm{dm}(\mathrm{run}(\sigma)) \mid \sigma$ is an accepting transition sequence on $G_r\}$ by definition of G_r and the run predicate. As, by Lemma 66, $L(\overline{r}) = \{\mathrm{run}(\sigma) \mid \sigma$ is an accepting transition sequence on $G_r\}$, we hence obtain $L(r) = L(G_r)$.

We next turn to the second statement: r is strongly deterministic if and only if G_r is deterministic. For the right to left direction, suppose r is not strongly deterministic, we show that then G_r is not deterministic. Here, r can be not strongly deterministic for two reasons: either it is not weakly deterministic, or it is but violates the second criterion in Definition 52.

First, suppose r is not weakly deterministic, and hence there exists words $\overline{u}, \overline{v}, \overline{w}$ and symbols $\overline{x}, \overline{y}$, with $\overline{x} \neq \overline{y}$ but $\mathrm{dm}(\overline{x}) = \mathrm{dm}(\overline{y})$ such that $\overline{uxv} \in L(\overline{r})$, and $\overline{uyw} \in L(\overline{r})$. Then, by Lemma 66, there exist accepting transition sequences $\sigma_1 = t_1, \ldots, t_n$ and $\sigma_2 = t'_1, \ldots, t'_m$ such that $\mathrm{run}(\sigma_1) = \overline{uxv}$ and $\mathrm{run}(\sigma_2) = \overline{uyw}$. Let $|\overline{u}| = i$. Then, $t_{i+1} \neq t'_{i+1}$, as t_{i+1} is a transition to $q_{\overline{x}}$, and t'_{i+1} to $q_{\overline{y}}$, with $q_{\overline{x}} \neq q_{\overline{y}}$. Let $j \in [1, i+1]$ be the smallest number such that $t_j \neq t'_j$, which must exist as $t_{i+1} \neq t'_{i+1}$. Let $u = \mathrm{dm}(\overline{ux})$. Then, after reading the prefix of u of length $j-1$, we have that G_r can be in some configuration γ and can both follow transition t_j and t'_j while reading the jth symbol of u. Hence, G_r is not deterministic.

Second, suppose r is weakly deterministic, but there exist words $\widetilde{u}, \widetilde{v}, \widetilde{w}$ over $\Sigma \cup \Gamma$, words $\alpha \neq \beta$ over Γ, and symbol $a \in \Sigma$ such that $\widetilde{u}\alpha a\widetilde{v}$ and $\widetilde{u}\beta a\widetilde{w}$ are correctly bracketed and in $L(\widetilde{r})$. Consequently, there exist words $\widetilde{\overline{u}}$, $\widetilde{\overline{v}}$, and $\widetilde{\overline{w}}$ and $\overline{x}$ such that $\widetilde{\overline{u}}\alpha\overline{x}\widetilde{\overline{v}}$ and $\widetilde{\overline{u}}\beta\overline{x}\widetilde{\overline{w}}$ are correctly bracketed and in $L(\widetilde{\overline{r}})$. Here, both words must use the same marked symbol $\overline{x}$ for a as r is weakly deterministic. Then, by Lemma 66, there exist transition sequences $\sigma_1 = t_1, \ldots, t_n$, $\sigma_2 = t_1, \ldots, t'_n$ such that $\text{brack-seq}(\sigma_1) = \widetilde{\overline{u}}\alpha\overline{x}\widetilde{\overline{v}}$ and $\text{brack-seq}(\sigma_2) = \widetilde{\overline{u}}\beta\overline{x}\widetilde{\overline{w}}$. Without loss of generality we can assume that α and β are chosen to be maximal (i.e. $\widetilde{\overline{u}}$ is either empty or ends with a marked symbol). But then, let $i = |\mathrm{strip}(\widetilde{\overline{u}})|$ and observe that $\mathrm{word}(t_{i+1}) = \alpha\overline{x}$ and $\mathrm{word}(t'_{i+1}) = \beta\overline{x}$, and thus as $\mathrm{word}(t_{i+1}) \neq \mathrm{word}(t'_{i+1})$ also $t_{i+1} \neq t'_{i+1}$. By reasoning exactly as in the previous case, it then follows that G_r is not deterministic.

Before proving the converse direction, we first make two observations. First, for any two transitions t and t', with $t \neq t'$ who share the same source state $q_{\overline{x}}$, we have that word$(t) \neq$ word(t'). Indeed, observe that if the target state of t is $q_{\overline{y}}$, and the target of t' is $q_{\overline{y}'}$, that then word$(t) = \alpha\overline{y}$ and word$(t') = \beta\overline{y}'$, with α, β words over Γ. Hence, if $\overline{y} \neq \overline{y}'$, we immediately have word$(t) \neq$ word(t'). Otherwise, when $\overline{y} = \overline{y}'$, we know that t is computed based on either (1) $(\overline{x}, \overline{y}, \text{null}) \in \text{follow}(\overline{r})$ and t' on $(\overline{x}, \overline{y}, c) \in \text{follow}(\overline{r})$ or (2) $(\overline{x}, \overline{y}, c) \in \text{follow}(\overline{r})$ and t' on $(\overline{x}, \overline{y}, c') \in \text{follow}(\overline{r})$, with $c \neq c'$. In both cases it is easily seen that $\alpha \neq \beta$. The second observation we make is that G_r is *reduced*, i.e., for every reachable configuration γ, there is some string which brings γ to an accepting configuration. The latter is due to the upper bound tests present in G_r.

Now, assume that G_r is not deterministic. We show that r is not strongly deterministic. As G_r is reduced and not deterministic there exist words u, v, w and symbol a such that $uav, uaw \in L(G_r)$ witnessed by accepting transition sequences $\sigma_1 = t_1, \ldots, t_n$ and $\sigma_2 = t'_1, \ldots, t'_m$ (for uav and uaw, respectively), such that $t_i = t'_i$, for all $i \in [1, |u|]$, but $t_{|u|+1} \neq t'_{|u|+1}$. Let $q_{\overline{x}}$ be the target of transition $t_{|u|+1}$ and $q_{\overline{y}}$ the target of transition $t'_{|u|+1}$, and $\overline{x}$ and $\overline{y}$ their associated marked symbols. Note that $\text{dm}(\overline{x}) = \text{dm}(\overline{y}) = a$. We now distinguish two cases.

First, assume $\overline{x} \neq \overline{y}$. By Lemma 66, both run$(\sigma_1) \in L(\overline{r})$ and run$(\sigma_2) \in L(\overline{r})$. Writing run$(\sigma_1) = \overline{uxv}$ and run$(\sigma_2) = \overline{uyw}$ (such that $|u| = |\overline{u}|$, $|v| = |\overline{v}|$ and $|w| = |\overline{w}|$) we then obtain the strings $\overline{u}$, $\overline{v}$, $\overline{w}$ and symbols $\overline{x}, \overline{y}$ sufficient to show that r is not weakly deterministic, and hence also not strongly deterministic.

Second, assume $\overline{x} = \overline{y}$. We then consider brack-seq(σ_1) and brack-seq(σ_2) which, by Lemma 66, are both correctly bracketed words in $L(\widetilde{\overline{r}})$. Further, note that $t_{|u|+1}$ and $t'_{|u|+1}$ share the same source state, and hence, by the first observation above, word$(t_{|u|+1}) = \alpha\overline{x} \neq$ word$(t'_{|u|+1}) = \beta\overline{y}$. In particular, as $\overline{x} = \overline{y}$, it holds that $\alpha \neq \beta$. But then, writing brack-seq$(\sigma_1) = \widetilde{\overline{u}}\alpha\overline{x}\widetilde{\overline{v}}$, and brack-seq$(\sigma_2) = \widetilde{\overline{u}}\alpha\overline{y}\widetilde{\overline{w}}$, we can see that $\text{dm}(\widetilde{\overline{u}})$, $\text{dm}(\widetilde{\overline{v}})$, $\text{dm}(\widetilde{\overline{w}})$, α, β, and a, violate the condition in Definition 52 and hence show that r is not strongly deterministic. $\square$

6.6 Testing Strong Determinism

Definition 52, defining strong determinism, is of a semantic 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 [66] give a cubic algorithm for RE(#), while Brüggemann-Klein [15] gives a quadratic algorithm for RE by computing its Glushkov

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

```
    $\overline{r} \leftarrow$ marked version of $r$
    Initialize Follow $\leftarrow \emptyset$
    Compute first($\overline{s}$), last($\overline{s}$), for all subexpressions $\overline{s}$ of $\overline{r}$
    **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**
        **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**
            $F \leftarrow \{(\overline{x}, \mathrm{dm}(\overline{y})) \mid \overline{x} \in \mathrm{last}(\overline{s}_1), \overline{y} \in \mathrm{first}(\overline{s}_2)\}$
        **else if** $\overline{s} = \overline{s}_1^{[k,\ell]}$, with $\ell \geq 2$ **then**
            **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**
            $F \leftarrow \{(\overline{x}, \mathrm{dm}(\overline{y})) \mid \overline{x} \in \mathrm{last}(\overline{s}_1), \overline{y} \in \mathrm{first}(\overline{s}_1)\}$
        **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**
            Follow $\leftarrow$ Follow $\uplus F$
    **return true**
```

automaton and testing whether it is deterministic[5].

We next show that Algorithm 1 is correct. Recall that we write $\overline{s} \preceq \overline{r}$ when $\overline{s}$ is a subexpression of $\overline{r}$ and $\overline{s} \prec \overline{r}$ when $\overline{s} \preceq \overline{r}$ and $\overline{s} \neq \overline{r}$.

Theorem 68. For any $r \in \mathrm{RE}(\#)$, isStrongDeterministic (r) returns true if and only if r is strongly deterministic. Moreover, it runs in time $\mathcal{O}(|r|^3)$.

Proof. Let $r \in \mathrm{RE}(\#)$, $\overline{r}$ a marking of r, and G_r the corresponding Glushkov counting automaton. By Theorem 65 it suffices to show that the algorithm isStrongDeterministic (r) returns true if and only if G_r is deterministic. Thereto, we first extract from Algorithm 1 the reasons for isStrongDeterministic (r) to return false. This is the case if and only if there exist marked symbols $\overline{x}, \overline{y}, \overline{y}'$, with $\mathrm{dm}(\overline{y}) = \mathrm{dm}(\overline{y}')$, such that either

1. $\overline{y}, \overline{y}' \in \mathrm{first}(\overline{s})$ and $\overline{y} \neq \overline{y}'$ (Line 4)
2. $(\overline{x}, \overline{y}, c) \in \mathrm{follow}(\overline{s})$, $(\overline{x}, \overline{y}', c) \in \mathrm{follow}(\overline{s})$, $\overline{y} \neq \overline{y}'$ and $\mathrm{upper}(c) \geq 2$ (Line 10)

[5]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.

3. $(\overline{x}, \overline{y}, c) \in \text{follow}(\overline{s})$, $(\overline{x}, \overline{y}', c') \in \text{follow}(\overline{s})$, $c \prec c'$, and $\text{upper}(c) \geq 2$, $\text{upper}(c') \geq 2$, and $\text{upper}(c) > \text{lower}(c)$ (Line 12)

4. $(\overline{x}, \overline{y}, \text{null}) \in \text{follow}(\overline{s})$, $(\overline{x}, \overline{y}', c') \in \text{follow}(\overline{s})$, $\text{lca}(\overline{x}, \overline{y}) \prec c'$ and $\text{upper}(c') \geq 2$ (Line 12)

5. $(\overline{x}, \overline{y}, c) \in \text{follow}(\overline{s})$, $(\overline{x}, \overline{y}', \text{null}) \in \text{follow}(\overline{s})$, $c \prec \text{lca}(\overline{x}, \overline{y}')$, $\text{upper}(c) \geq 2$, and $\text{upper}(c) > \text{lower}(c)$ (Line 12)

6. $(\overline{x}, \overline{y}, \text{null}) \in \text{follow}(\overline{s})$, $(\overline{x}, \overline{y}', \text{null}) \in \text{follow}(\overline{s})$, $\overline{y} \neq \overline{y}'$ and $\text{lca}(\overline{x}, \overline{y}) = \text{lca}(\overline{x}, \overline{y}')$ (Line 7)

7. $(\overline{x}, \overline{y}, \text{null}) \in \text{follow}(\overline{s})$, $(\overline{x}, \overline{y}', \text{null}) \in \text{follow}(\overline{s})$, and $\text{lca}(\overline{x}, \overline{y}) \prec \text{lca}(\overline{x}, \overline{y}')$ (Line 12)

We now show that G_s is not deterministic if and only if one of the above seven conditions holds. We first verify the right to left direction by investigating the different cases.

Suppose case 1 holds, i.e., $\overline{y}, \overline{y}' \in \text{first}(\overline{s})$ and $\overline{y} \neq \overline{y}'$. Then, there is a transition from q_0 to $q_{\overline{y}}$ and one from q_0 to $q_{\overline{y}'}$, with $q_{\overline{y}} \neq q_{\overline{y}'}$. These transitions can both be followed when the first symbol in a string is $\text{dm}(\overline{y})$. Hence, G_s is not deterministic.

In each of the six remaining there are always two distinct tuples in $\text{follow}(\overline{s})$ which generate distinct transitions with as source state $q_{\overline{x}}$ and target states $q_{\overline{y}}$ and $q_{\overline{y}'}$, and $\text{dm}(\overline{y}) = \text{dm}(\overline{y}')$. Therefore, it suffices, in each of the cases, to construct a reachable configuration $\gamma = (q_{\overline{x}}, \alpha)$ from which both transitions can be followed by reading $\text{dm}(\overline{y})$. The reachability of this configuration γ will always follow from Lemma 69, but its precise form depends on the particular case we are in. In each of the following cases, let $\text{iterators}(\overline{x}) = [c_1, \ldots, c_n]$ and, when applicable, set $c = c_i$ and $c' = c_j$. When both c and c' occur, we always have $c \prec c'$, and hence $i < j$.

Case 2: Set $\gamma = (q_{\overline{x}}, \alpha)$ with $\alpha(\text{cv}(c_m)) = \text{lower}(c_m)$, for all $m \in [1, i-1]$, and $\alpha(\text{cv}(c_m)) = 1$, for all $m \in [i, m]$. We need to show that the transitions generated by $(\overline{x}, \overline{y}, c)$ and $(\overline{x}, \overline{y}', c)$ can both be followed from γ. This is due to the fact that $\alpha \models \text{value-test}_{[c_1,\ldots,c_{i-1}]}$ and $\alpha \models \text{upperbound-test}_{[c_i]}$. The latter because $\text{upper}(c_i) \geq 2 > \alpha(\text{cv}(c_i))$.

Case 3: Set $\gamma = (q_{\overline{x}}, \alpha)$ with $\alpha(\text{cv}(c_m)) = \text{lower}(c_m)$, for all $m \in [1, j-1]$, and $\alpha(\text{cv}(c_m)) = 1$, for all $m \in [j, n]$. To see that the transition generated by $(\overline{x}, \overline{y}, c)$ can be followed, note that again $\alpha \models \text{value-test}_{[c_1,\ldots,c_{i-1}]}$ and $\alpha \models \text{upperbound-test}_{[c_i]}$. The latter due to the fact that $\text{upper}(c) > \text{lower}(c) = \alpha(\text{cv}(c_i))$. On the other hand, also $\alpha \models \text{value-test}_{[c_1,\ldots,c_{j-1}]}$ and $\alpha \models \text{upperbound-test}_{[c_j]}$ as $\text{upper}(c') \geq 2 > \alpha(\text{cv}(c_j))$. Hence, G_s can also follow the transition generated by $(\overline{x}, \overline{y}', c')$.

Case 4: Set $\gamma = (q_{\overline{x}}, \alpha)$ with $\alpha(\text{cv}(c_m)) = \text{lower}(c_m)$, for all $m \in [1, j-1]$, and $\alpha(\text{cv}(c_m)) = 1$, for all $m \in [j, n]$. Arguing as above, and using the facts that (1) $\text{upper}(c') \geq 2$ and (2) $\text{iterators}(\overline{x}, \overline{y}) = [c_1, \ldots, c_{i'}]$, for some $i' < j$ (as $\text{lca}(\overline{x}, \overline{y}) \prec c'$); we can deduce that G_s can again follow both transitions.

Cases 5, 6, and 7: In each of these cases we can set $\gamma = (q_{\overline{x}}, \alpha)$ with $\alpha(\text{cv}(c_m)) = \text{lower}(c_m)$, for all $m \in [1, n]$. Arguing as before it can be seen that in each case both transitions can be followed from this configuration.

We next turn to the left to right direction. Assume G_s is not deterministic and let γ be a reachable configuration such that two distinct transitions t_1, t_2 can be followed by reading a symbol a. We argue that in this case the conditions for one of the seven cases above must be fulfilled. First, if $\gamma = (q_0, \alpha)$ then t_1 and t_2 must be the initial transitions of a run, as there are no transitions returning to q_0. As, furthermore, q_0 has exactly one transition to each state $q_{\overline{x}}$, when $\overline{x} \in \text{first}(\overline{s})$, we can see that the conditions in case 1 hold.

Therefore, assume $\gamma = (q_{\overline{x}}, \alpha)$, with $\overline{x} \in \text{Char}(\overline{r})$. We will investigate the tuples in $\text{follow}(\overline{s})$ which generated t_1 and t_2 and show that they fall into one of the six remaining cases mentioned above, hence forcing isStrongDeterministic (s) to return false. There are indeed six possibilities, ignoring symmetries, which we immediately classify to the case they will belong to:

2. t_1 is generated by $(\overline{x}, \overline{y}, c)$ and t_2 by $(\overline{x}, \overline{y}', c)$,

3. t_1 is generated by $(\overline{x}, \overline{y}, c)$ and t_2 by $(\overline{x}, \overline{y}', c')$ with $c \prec c'$.

4. t_1 is generated by $(\overline{x}, \overline{y}, \text{null})$ and t_2 by $(\overline{x}, \overline{y}', c')$ with $\text{lca}(\overline{x}, \overline{y}) \prec c'$.

5. t_1 is generated by $(\overline{x}, \overline{y}, c)$ and t_2 by $(\overline{x}, \overline{y}', \text{null})$ and $c \prec \text{lca}(\overline{x}, \overline{y})$.

6. t_1 is generated by $(\overline{x}, \overline{y}, \text{null})$ and t_2 by $(\overline{x}, \overline{y}', \text{null})$ with $\text{lca}(\overline{x}, \overline{y}) = \text{lca}(\overline{x}, \overline{y}')$.

7. t_1 is generated by $(\overline{x}, \overline{y}, \text{null})$ and t_2 by $(\overline{x}, \overline{y}', \text{null})$ with $\text{lca}(\overline{x}, \overline{y}) \prec \text{lca}(\overline{x}, \overline{y}')$.

Note that all subexpressions under consideration (i.e. $\text{lca}(\overline{x}, \overline{y})$, $\text{lca}(\overline{x}, \overline{y}')$, c, and c') contain x, and that $\text{lca}(\overline{x}, \overline{y})$ and $\text{lca}(\overline{x}, \overline{y}')$ are always subexpressions whose topmost operator is a concatenation. These are the reasons why all these expressions are in a subexpression relation, and why we never have, for instance, $\text{lca}(\overline{x}, \overline{y}) = c$.

However, these six cases only give us the different possibilities of how the transitions t_1 and t_2 can be generated by tuples in $\text{follow}(\overline{s})$. We still need to argue that the additional conditions imposed by the cases mentioned above apply. Thereto, we first note that for all iterators c and c' under consideration,

$\text{upper}(c) \geq 2$ and $\text{upper}(c') \geq 2$ must surely hold. Indeed, suppose for instance $\text{upper}(c) = 1$. Then, for any transition t generated by $(\overline{x}, \overline{y}, c)$ the guard ϕ contains the condition $\text{upperbound-test}_c := \text{cv}(c) < 1$, which can never be true. Hence, such a transition t can never be followed and is not relevant. This already shows that possibilities 4 and 7 above, indeed imply cases 4 and 7, respectively. For the additional cases, we argue on a case by case basis.

Cases 2 and 6: We additionally need to show $\overline{y} \neq \overline{y}'$, which is immediate from the fact that $t_1 \neq t_2$. Indeed, assuming $\overline{y} = \overline{y}'$, implies that t_1 and t_2 are generated by the same tuple in $\text{follow}(\overline{s})$ and are hence equal.

Cases 3 and 5: We need to show $\text{upper}(c) > \text{lower}(c)$. In both cases, the guard of transition t_1 contains the condition $\text{cv}(c) < \text{upper}(c)$ as upperbound test, whereas the guard of transition t_2 contains the condition $\text{cv}(c) \geq \text{lower}(c)$. These can only simultaneously be true when $\text{upper}(c) > \text{lower}(c)$.

This settles the correctness of the algorithm. We conclude by arguing that the algorithm runs in time $\mathcal{O}(|r|^3)$. Computing the first and last sets for each subexpression $\overline{s}$ of $\overline{r}$ can easily be done in time $\mathcal{O}(|r|^3)$ as can the test on Line 4. Further, the for loop iterates over a linear number of nodes in the parse tree of $\overline{r}$. To do each iteration of the loop in quadratic time, one needs to implement the set Follow as a two-dimensional boolean table. In each iteration we then need to do at most a quadratic number of (constant time) lookups and writes to the table. Altogether this yields a quadratic algorithm. □

It only remains to prove the following lemma:

Lemma 69. Let $\overline{x} \in \text{Char}(\overline{r})$, and $\text{iterators}(\overline{x}) = [c_1, \ldots, c_n]$. Let $\gamma = (q_{\overline{x}}, \alpha)$, be a configuration such that $\alpha(\text{cv}(c_i)) \in [1, \text{upper}(c_i)]$, for $i \in [1, n]$, and $\alpha(\text{cv}(c)) = 1$, for all other countervariables c. Then, γ is reachable in G_r.

Proof. This lemma can easily be proved by induction on the structure of r. However, as this is a bit tedious, we only provide some intuition by constructing, given such a configuration γ, a string w which brings G_s from its initial configuration to γ.

We construct w by concatenating several substrings. Thereto, for every $i \in [1, n]$, let v_i be a non-empty string in $L(\text{base}(c_i))$. Concatenating such a string v_i with itself allows to iterate c_i. We further define, for every $i \in [1, n]$, a marked string $\overline{w}_i$ which, intuitively, connects the different iterators. Thereto, let $\overline{w}_n$ be a minimal (w.r.t. length) string such that $\overline{w}_n \in (\text{Char}(\overline{s}) \setminus \text{Char}(\overline{c_n}))^*$ and such that there exist $\overline{u} \in \text{Char}(\overline{c_n})^*$ and $\overline{v} \in \text{Char}(\overline{s})^*$ such that $\overline{w}_n \overline{u} \overline{x} \overline{v} \in L(\overline{s})$. Similarly, for any $i \in [1, n-1]$, let $\overline{w}_i$ be a minimal (w.r.t. length) string such that $\overline{w}_i \in (\text{Char}(\overline{c_{i+1}}) \setminus \text{Char}(\overline{c_i}))^*$ and there exist $\overline{u} \in \text{Char}(\overline{c_i})^*$ and $\overline{v} \in \text{Char}(\overline{c_{i+1}})^*$ such that $\overline{w}_n \overline{u} \overline{x} \overline{v} \in L(\overline{c_{i+1}})$. Finally, let $\overline{w}_0$ be a string such that there exists a $\overline{u}$ such that $\overline{w}_0 \overline{x} \overline{u} \in L(\text{base}(c_1))$. We require these strings

$\overline{w}_1$ to $\overline{w}_n$ to be minimal to be sure that they do not allow to iterate over their corresponding counter.

Then, the desired string w is

$$\mathrm{dm}(\overline{w}_n)(v_n)^{\alpha(\mathrm{cv}(c_n))}\mathrm{dm}(\overline{w}_{n-1})(v_{n-1})^{\alpha(\mathrm{cv}(c_{n-1}))}\cdots\mathrm{dm}(\overline{w}_0)\mathrm{dm}(\overline{x})\,.$$

□

6.7 Decision Problems for RE(#)

We recall the following complexity results:

Theorem 70. 1. INCLUSION and EQUIVALENCE for RE(#) are EXPSPACE-complete [83], INTERSECTION for RE(#) is PSPACE-complete (Chapter 5)

2. INCLUSION and EQUIVALENCE for DRE_W are in PTIME, INTERSECTION for DRE_W is PSPACE-complete [75].

3. INCLUSION for $\mathrm{DRE}^{\#}_W$ is coNP-hard [64].

By combining (1) and (2) of Theorem 70 we get the complexity of INTERSECTION for $\mathrm{DRE}^{\#}_W$ and $\mathrm{DRE}^{\#}_S$. This is not the case for the INCLUSION and EQUIVALENCE problem, unfortunately. By using the results of the previous sections we can, for $\mathrm{DRE}^{\#}_S$, give a PSPACE upperbound for both problems, however.

Theorem 71. (1) EQUIVALENCE and INCLUSION for $\mathrm{DRE}^{\#}_S$ are in PSPACE.

(2) INTERSECTION for $\mathrm{DRE}^{\#}_W$ and $\mathrm{DRE}^{\#}_S$ is PSPACE-complete.

Proof. (1) We show that INCLUSION for $\mathrm{DRE}^{\#}_S$ is in PSPACE. Given two strongly deterministic RE(#) expressions r_1, r_2, we construct two CDFAs A_1, A_2 for r_1 and r_2 using the construction of section 6.5, which by Theorem 65 are indeed deterministic. Then, we construct the CDFAs A_2', A such that A_2' accepts the complement of A_2, and A is the intersection of A_1 and A_2'. This can all be done in polynomial time and preserves determinism according to Theorem 64. Finally, $L(r_1) \subseteq L(r_2)$ if and only if $L(A) \neq \emptyset$, which can be decided using only polynomial space by Theorem 64(1).

(2) The result for $\mathrm{DRE}^{\#}_W$ is immediate from Theorem 70(1) and (2). This result also carries over to $\mathrm{DRE}^{\#}_S$. For the upper bound this is trivial, whereas the lower bound follows from the fact that standard weakly deterministic regular expressions can be transformed in linear time into equivalent strongly deterministic expressions [15]. □

6.8 Discussion

We investigated and compared the notions of strong and weak determinism for regular expressions 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:

- Is it decidable if a language is definable by a weakly deterministic expression with counting? For standard (weakly) deterministic expressions, a decision algorithm is given by Bruggemann-Klein and Wood [17] which, by our results, also carries over to strongly deterministic expressions with counting.

- Can the Glushkov construction given in Section 6.5 be extended such that it translates any weakly deterministic expression with counting into a CDFA? Kilpelainen and Tuhkanen [65] have given a Glushkov-like construction translating weakly deterministic expressions into a deterministic automata model. However, this automaton model is mostly designed for doing membership testing and complexity bounds can not directly be derived from it, due to the lack of upperbounds on the counters.

- What are the exact complexity bounds for inclusion and equivalence of strongly and weakly deterministic expressions with counting? For strong determinism, this can still be anything between PTIME and PSPACE, whereas for weak determinism, coNP and PSPACE are the most likely, although EXPSPACE also remains possible.

Part II

Applications to XML Schema Languages

7 Succinctness of Pattern-Based Schema Languages

In formal language theoretic terms, an XML schema defines a tree language. The for historical reasons still widespread Document Type Definitions (DTDs) can then be seen as context-free grammars with regular expressions at right-hand sides which define the local tree languages [16]. XML Schema [100] extends the expressiveness of DTDs by a typing mechanism allowing content-models to depend on the type rather than only on the label of the parent. Unrestricted application of such typing leads to the robust class of unranked regular tree languages [16] as embodied in the XML schema language Relax NG [22]. The latter language is commonly abstracted in the literature by extended DTDs (EDTDs) [88]. The Element Declarations Consistent constraint in the XML Schema specification, however, restricts this typing: it forbids the occurrence of different types of the same element in the same content model. Murata et al. [85] therefore abstracted XSDs by single-type EDTDs. Martens et al. [77] subsequently characterized the expressiveness of single-type EDTDs in several syntactic and semantic ways. Among them, they defined an extension of DTDs equivalent in expressiveness to single-type EDTDs: ancestor-guarded DTDs. An advantage of this language is that it makes the expressiveness of XSDs more apparent: the content model of an element can only depend on regular string properties of the string formed by the ancestors of that element. Ancestor-based DTDs can therefore be used as a type-free front-end for XML Schema. As they can be interpreted both

	other semantics	EDTD	EDTD$^{\mathrm{st}}$	DTD
$\mathcal{P}_\exists$(Reg)	2-exp (83(1))	exp (83(2))	exp (83(3))	exp* (83(5))
$\mathcal{P}_\forall$(Reg)	2-exp (83(6))	2-exp (83(7))	2-exp (83(8))	2-exp (83(10))
$\mathcal{P}_\exists$(Lin)	\ (85(1))	exp (85(2))	exp (85(3))	exp* (85(5))
$\mathcal{P}_\forall$(Lin)	\ (85(6))	2-exp (85(7))	2-exp (85(8))	2-exp (85(10))
$\mathcal{P}_\exists$(S-Lin)	poly (89(1))	poly (89(2))	poly (89(3))	poly (89(6))
$\mathcal{P}_\forall$(S-Lin)	poly (89(7))	poly (89(8))	poly (89(9))	poly (89(12))
$\mathcal{P}_\exists$(Det-S-Lin)	poly (89(1))	poly (89(2))	poly (89(3))	poly (89(6))
$\mathcal{P}_\forall$(Det-S-Lin)	poly (89(7))	poly (89(8))	poly (89(9))	poly (89(12))

Table 7.1: Overview of complexity results for translating pattern-based schemas into other schema formalisms. For all non-polynomial complexities, except the ones marked with a star, there exist examples matching this upper bound. Theorem numbers are given between brackets.

in an existential and universal way, we study in this chapter the complexity of translating between the two semantics and into the formalisms of DTDs, EDTDs, and single-type EDTDs.

In the remainder of the chapter, we use the name pattern-based schema, rather than ancestor-based DTD, as it emphasizes the dependence on a particular pattern language. A pattern-based schema is a set of rules of the form (r, s), where r and s are regular expressions. An XML tree is then existentially valid with respect to a rule set if for each node there is a rule such that the path from the root to that node matches r and the child sequence matches s. Furthermore, it is universally valid if each node vertically matching r, horizontally matches s. The existential semantics is exhaustive, fully specifying every allowed combination, and more DTD-like, whereas the universal semantics is more liberal, enforcing constraints only where necessary.

Kasneci and Schwentick [62] studied the complexity of the satisfiability and inclusion problem for pattern-based schemas under the existential ($\exists$) and universal ($\forall$) semantics. They considered regular (Reg), linear (Lin), and strongly linear (S-Lin) patterns. These correspond to the regular expressions, XPath expressions with only child (`/`) and descendant (`//`), and XPath expressions of the form $//w$ or $/w$, respectively. Deterministic strongly linear (Det-S-Lin) patterns are strongly linear patterns in which additionally all horizontal expressions s are required to be deterministic [17]. A snapshot of their results is given in the third and fourth column of Table 7.2. These results indicate that there is no difference between the existential and universal semantics.

We, however, show that with respect to succinctness there is a huge difference. Our results are summarized in Table 7.1. Both for the pattern languages

	SIMPLIFICATION	SATISFIABILITY	INCLUSION
$\mathcal{P}_\exists$(Reg)	EXPTIME (83(4))	EXPTIME [62]	EXPTIME [62]
$\mathcal{P}_\forall$(Reg)	EXPTIME (83(9))	EXPTIME [62]	EXPTIME [62]
$\mathcal{P}_\exists$(Lin)	PSPACE (85(4))	PSPACE [62]	PSPACE [62]
$\mathcal{P}_\forall$(Lin)	PSPACE (85(9))	PSPACE [62]	PSPACE [62]
$\mathcal{P}_\exists$(S-Lin)	PSPACE (89(4))	PSPACE [62]	PSPACE [62]
$\mathcal{P}_\forall$(S-Lin)	PSPACE (89(10))	PSPACE [62]	PSPACE [62]
$\mathcal{P}_\exists$(Det-S-Lin)	in PTIME (89(5))	in PTIME [62]	in PTIME [62]
$\mathcal{P}_\forall$(Det-S-Lin)	in PTIME (89(11))	in PTIME [62]	in PTIME [62]

Table 7.2: Overview of complexity results for pattern-based schemas. All results, unless indicated otherwise, are completeness results. Theorem numbers for the new results are given between brackets.

Reg and Lin, the universal semantics is exponentially more succinct than the existential one when translating into (single-type) extended DTDs and ordinary DTDs. Furthermore, our results show that the general class of pattern-based schemas is ill-suited to serve as a front-end for XML Schema due to the inherent exponential or double exponential size increase after translation. Only when resorting to S-Lin patterns, there are translations only requiring polynomial size increase. Fortunately, the practical study in [77] shows that the sort of typing used in XSDs occurring in practice can be described by such patterns. Our results further show that the expressive power of the existential and the universal semantics coincide for Reg and S-Lin, albeit a translation can not avoid a double exponential size increase in general in the former case. For linear patterns the expressiveness is incomparable. Finally, as listed in Table 7.2, we study the complexity of the simplification problem: given a pattern-based schema, is it equivalent to a DTD?

Outline. In **Section 7.1**, we introduce some additional definitions concerning schema languages, and pattern-based schemas. In **Section 7.2**, **7.3**, and **7.4**, we study pattern-based schemas with regular, linear, and strongly linear expressions, respectively.

7.1 Preliminaries and Basic Results

In this section, we introduce pattern-based schemas, an alternative characterization of single-type EDTDs, recall some known theorems, and introduce some additional terminology for describing succinctness results.

7.1.1 Pattern-Based Schema Languages

We recycle the following definitions from [62].

Definition 72. A *pattern-based schema* P is a set $\{(r_1, s_1), \ldots, (r_m, s_m)\}$ where all r_i, s_i are regular expressions.

Each pair (r_i, s_i) of a pattern-based schema represents a schema rule. We also refer to the r_i and s_i as the vertical and horizontal regular expressions, respectively. There are two semantics for pattern-based schemas.

Definition 73. A tree t is *existentially valid* with respect to a pattern-based schema P if, for every node v of t, there is a rule $(r, s) \in P$ such that $\text{anc-str}(v) \in L(r)$ and $\text{ch-str}(v) \in L(s)$. In this case, we write $P \models_\exists t$.

Definition 74. A tree t is *universally valid* with respect to a pattern-based schema P if, for every node v of t, and each rule $(r, s) \in P$ it holds that $\text{anc-str}(v) \in L(r)$ implies $\text{ch-str}(v) \in L(s)$. In this case, we write $P \models_\forall t$.

Denote by $P_\exists(t) = \{v \in \text{Dom}(t) \mid \exists(r, s) \in P, \text{anc-str}(v) \in L(r) \wedge \text{ch-str}(v) \in L(s)\}$ the set of nodes in t that are existentially valid. Denote by $P_\forall(t) = \{v \in \text{Dom}(t) \mid \forall(r, s) \in P, \text{anc-str}(v) \in L(r) \Rightarrow \text{ch-str}(v) \in L(s)\}$ the set of nodes in t that are universally valid.

We denote the set of Σ-trees which are existentially and universally valid with respect to P by $\mathcal{T}_\exists^\Sigma(P)$ and $\mathcal{T}_\forall^\Sigma(P)$, respectively. We often omit Σ if it is clear from the context what the alphabet is.

When for every string $w \in \Sigma^*$ there is a rule $(r, s) \in P$ such that $w \in L(r)$, then we say that P is *complete*. Further, when for every pair $(r, s), (r', s') \in P$ of different rules, $L(r) \cap L(r') = \emptyset$, then we say that P is *disjoint*.

In some proofs, we make use of unary trees, which can be represented as strings. In this context, we abuse notation and write for instance $w \in \mathcal{T}_\exists(P)$ meaning that the unary tree which w represents is existentially valid with respect to P. Similarly, we refer to the last position of w as the leaf of w.

We conclude this section with two basic results on pattern-based schemas.

Lemma 75. For a pattern-based schema P, a tree t and a string w,

1. $t \in \mathcal{T}_\forall(P)$ if and only if for every node v of t, $v \in P_\forall(t)$.
2. if $w \in \mathcal{T}_\forall(P)$ then for every prefix w' of w and every non-leaf node v of w', $v \in P_\forall(w')$.
3. $t \in \mathcal{T}_\exists(P)$ if and only if for every node v of t, $v \in P_\exists(t)$.
4. if $w \in \mathcal{T}_\exists(P)$ then for every prefix w' of w and every non-leaf node v of w', $v \in P_\exists(w')$.

Proof. (1,3) These are in fact just a restatement of the definition of universal and existential satisfaction and are therefore trivially true.

(2) Consider any non-leaf node v' of w'. Since w' is a prefix of w, there must be a node v of w such that $\text{anc-str}^{w}(v) = \text{anc-str}^{w'}(v')$ and $\text{ch-str}^{w}(v) = \text{ch-str}^{w'}(v')$. By Lemma 75(1), $v \in P_{\forall}(w)$ and thus $v' \in P_{\forall}(w)$.

(4) The proof of (2) carries over literally for the existential semantics. □

Lemma 76. For any complete and disjoint pattern-based schema P, $\mathcal{T}_{\exists}(P) = \mathcal{T}_{\forall}(P)$.

Proof. We show that if P is complete and disjoint, then for any node v of any tree t, $v \in P_{\exists}(t)$ if and only if $v \in P_{\forall}(t)$. The lemma then follows from Lemma 75(1) and (3). First, suppose $v \in P_{\exists}(t)$. Then, there is a rule $(r, s) \in P$ such that $\text{anc-str}(v) \in L(r)$ and $\text{ch-str}(v) \in L(s)$, and by the disjointness of P, $\text{anc-str}(v) \notin L(r')$ for any other vertical expression r' in P. It thus follows that $v \in P_{\forall}(t)$. Conversely, suppose $v \in P_{\forall}(t)$. By the completeness of P there is at least one rule (r, s) such that $\text{anc-str}(v) \in L(r)$ and thus $\text{ch-str}(v) \in L(s)$. It follows that $v \in P_{\exists}(t)$. □

7.1.2 Characterizations of (Extended) DTDs

We start by introducing another schema formalism equivalent to single-type EDTDs. An *automaton-based schema* D over vocabulary Σ is a tuple (A, λ), where $A = (Q, q_0, \delta, F)$ is a DFA and λ is a function mapping states of A to regular expressions. A tree t is accepted by D if for every node v of t, where $q \in Q$ is the state such that $q_0 \Rightarrow_{A,\text{anc-str}(v)} q$, $\text{ch-str}(v) \in L(\lambda(q))$. Because the set of final states F of A is not used, we often omit F and represent A as a triple (Q, q_0, δ). Automaton-based schemas are implicit in [77] and introduced explicitly in [76].

Remark 77. Because DTDs and EDTDs only define tree languages in which every tree has the same root element, we implicitly assume that this is also the case for automaton-based schemas and the pattern-based schemas defined next. Whenever we translate among pattern-based schemas, we drop this assumption. Obviously, this does not influence any of the results of this chapter.

Lemma 78. Any automaton-based schema D can be translated into an equivalent single-type EDTD D' in time at most quadratic in the size of D, and vice versa.

Proof. Let $D = (A, \lambda)$, with $A = (Q, q_0, \delta)$, be an automaton-based schema. We start by making A complete. That is, we add a sink state $q_{\_}$ to Q and for every pair $q \in Q$, $a \in \Sigma$, for which there is no transition $(q, a, q') \in \delta$,

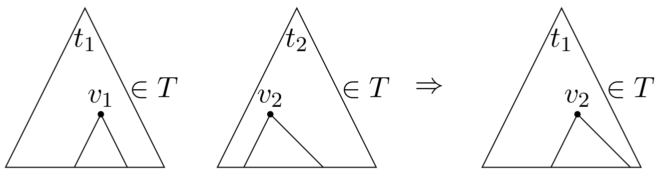


Figure 7.1: Closure under label-guarded subtree exchange

we add $(q, a, q_\_)$ to δ. Further, $\lambda(q_\_) = \emptyset$. Construct $D' = (\Sigma, \Sigma', d, s^i, \mu)$ as follows. Let s^i be such that s is the root symbol of any tree defined by D and $(q_0, s, q_i) \in \delta$. Let $Q \cup \{q_\_\} = \{q_0, \ldots, q_n\}$ for some $n \in \mathbb{N}$, then $\Sigma' = \{a^i \mid a \in \Sigma \wedge q_i \in Q\}$ and $\mu(a^i) = a$. Finally, $d(a^i) = \lambda(q_i)$, where any symbol $a \in \Sigma$ is replaced by a^j when $(q_i, a, q_j) \in \delta$. Since A is complete, a^j is guaranteed to exist and since A is a DFA a^j is uniquely defined. For the time complexity of the algorithm, we see that the number of types in D' can never be exceeded by the number of transitions in A. Then, to every type one regular expression from D' is assigned which yields a quadratic algorithm.

Conversely, let $D = (\Sigma, \Sigma', d, s, \mu)$ be a single-type EDTD. The equivalent automaton-based schema $D = (A, \lambda)$ with $A = (Q, q_0, \delta)$ is constructed as follows. Let $Q = \Sigma'$, $q_0 = s$, and for $a^i, b^j \in \Sigma'$, $(a^i, b, b^j) \in \delta$ if $\mu(b^j) = b$ and b^j occurs in $d(a^i)$. Note that since D is a single-type EDTD, A is guaranteed to be deterministic. Finally, for any type $a^i \in \Sigma'$, $\lambda(a^i) = \mu(d(a^i))$. □

A regular tree language $\mathcal{T}$ is closed under *label-guarded subtree exchange* if it has the following property: if two trees t_1 and t_2 are in $\mathcal{T}$, and there are two nodes v_1 in t_1 and v_2 in t_2 with the same label, then $t_1[v_1 \leftarrow \text{subtree}^{t_2}(v_2)]$ is also in $\mathcal{T}$. This notion is graphically illustrated in Figure 7.1.

Lemma 79 ([88])**.** A regular tree language is definable by a DTD if and only if it is closed under label-guarded subtree exchange.

An EDTD $D = (\Sigma, \Sigma', d, s_d, \mu)$ is *trimmed* if for for every $a^i \in \Sigma'$, there exists a tree $t \in L(d)$ and a node $u \in \text{Dom}(t)$ such that $\text{lab}^t(u) = a^i$.

Lemma 80 ([77])**.** 1. For every EDTD D, a trimmed EDTD D', with $L(D) = L(D')$, can be constructed in time polynomial in the size of D.

2. Let D be a trimmed EDTD. For any type $a^i \in \Sigma'$ and any string $w \in L(d(a^i))$ there exists a tree $t \in L(d)$ which contains a node v with $\text{lab}^t(v) = a^i$ and $\text{ch-str}^t(v) = w$.

7.1.3 Theorems

We use the following theorem of Glaister and Shallit [45].

Theorem 81 ([45]). Let $L \subseteq \Sigma^*$ be a regular language and suppose there exists a set of pairs $M = \{(x_i, w_i) \mid 1 \leq i \leq n\}$ such that

- $x_i w_i \in L$ for $1 \leq i \leq n$; and
- $x_i w_j \notin L$ for $1 \leq i, j \leq n$ and $i \neq j$.

Then any NFA accepting L has at least n states.

We make use of the following results on transformations of regular expressions.

Theorem 82.

1. Let $r_1, \ldots, r_n, s_1, \ldots, s_m$ be regular expressions. A regular expression r, with $L(r) = \bigcap_{i \leq n} L(r_i) \setminus \bigcup_{i \leq m} L(s_i)$, can be constructed in time double exponential in the sum of the sizes of all r_i, s_j, $i \leq n$, $j \leq m$.
2. For any regular expressions r and alphabet $\Delta \subseteq \Sigma$, an expression r^-, such that $L(r^-) = L(r) \cap \Delta^*$, can be constructed in time linear in the size of r.

Proof. (1) First, for every $i \leq n$, construct an NFA A_i, such that $L(r_i) = L(A_i)$, which can be done in polynomial time according to Theorem 2(4). Then, let A be the DFA accepting $\bigcap_{i \leq n} L(A_i)$ obtained from the A_i by determinization followed by a product construction. For k the size of the largest NFA, this can be done in time $\mathcal{O}(2^{k \cdot n})$. For every $i \leq m$, construct an NFA B_i, with $L(s_i) = B_i$, and let B_i be the DFA accepting $\bigcup_{i \leq m} L(B_i)$ again obtained from the B_i by means of determinization and a product construction. Similarly, B can also be computed in time exponential in the size of the input. Then, compute the DFA B' for the complement of B by making B complete and exchanging final and non-final states in B, which can be done in time polynomial in the size of B. Then, the DFA C accepts $L(A) \cap L(B')$ and can again be obtained by a product construction on A and B' which requires polynomial time in the sizes of A and B'. Therefore, C is of exponential size in function of the input. Finally, r, with $L(r) = \bigcap_{i \leq n} L(r_i) \setminus \bigcup_{i \leq m} L(s_i)$ can be obtained from C in time exponential in the size of C (Theorem 2(1)) and thus yields a double exponential algorithm in total.

(2) The algorithm proceeds in two steps. First, replace every symbol $a \notin \Delta$ in r by $\emptyset$. Then, use the following rewrite rules on subexpressions of r as often as possible: $\emptyset^* = \varepsilon$, $\emptyset s = s\emptyset = \emptyset$, and $\emptyset + s = s + \emptyset = s$. This gives us r^- which is equal to $\emptyset$ or does not contain $\emptyset$ at all, with $L(r^-) = L(r) \cap \Delta^*$. $\square$

7.1.4 Succinctness

As the focus in this chapter is on succinctness results we introduce some additional notation to allow for a succinct notation of such results.

For a class $\mathcal{S}$ and $\mathcal{S}'$ of representations of schema languages, and F a class of functions from $\mathbb{N}$ to $\mathbb{N}$, we write $\mathcal{S} \xrightarrow{F} \mathcal{S}'$ if there is an $f \in F$ such that for every $s \in \mathcal{S}$ there is an $s' \in \mathcal{S}'$ with $L(s) = L(s')$ which can be constructed in time $f(|s|)$. This also implies that $|s'| \leq f(|s|)$. By $L(s)$ we mean the set of trees defined by s.

We write $\mathcal{S} \overset{F}{\Rightarrow} \mathcal{S}'$ if $\mathcal{S} \xrightarrow{F} \mathcal{S}'$ and there is an $f \in F$, a monotonically increasing function $g : \mathbb{N} \to \mathbb{N}$ and an infinite family of schemas $s_n \in \mathcal{S}$ with $|s_n| \leq g(n)$ such that the smallest $s' \in \mathcal{S}'$ with $L(s) = L(s')$ is at least of size $f(g(n))$. By poly, exp and 2-exp we denote the classes of functions $\bigcup_{k,c} cn^k$, $\bigcup_{k,c} c2^{n^k}$ and $\bigcup_{k,c} c2^{2^{n^k}}$, respectively.

Further, we write $\mathcal{S} \not\rightarrow \mathcal{S}'$ if there exists an $s \in \mathcal{S}$ such that for every $s' \in \mathcal{S}'$, $L(s') \neq L(s)$. In this case we also write $\mathcal{S} \overset{F}{\not\rightarrow} \mathcal{S}'$ and $\mathcal{S} \overset{F}{\not\Rightarrow} \mathcal{S}'$ whenever $\mathcal{S} \xrightarrow{F} \mathcal{S}'$ and $\mathcal{S} \overset{F}{\Rightarrow} \mathcal{S}'$, respectively, hold for those elements in $\mathcal{S}$ which do have an equivalent element in $\mathcal{S}'$.

7.2 Regular Pattern-Based Schemas

In this section, we study the full class of pattern-based schemas which we denote by $\mathcal{P}_\exists(\text{Reg})$ and $\mathcal{P}_\forall(\text{Reg})$. The results are shown in Theorem 83. Notice that the translations among schemas with different semantics, and the translation from a pattern-based schema under universal semantics to an EDTD are double exponential, whereas the translation from a schema under existential semantics to an EDTD is “only” exponential. Essentially all these double exponential lower bounds are due to the fact that in these translations one necessarily has to apply operations, such as intersection and complement, on regular expressions, which, as shown in Chapter 3, yields double exponential lower bounds. In the translation from a pattern-based schema under existential semantics to an EDTD such operations are not necessary which allows for an easier translation.

Theorem 83. 1. $\mathcal{P}_\exists(\text{Reg}) \overset{\text{2-exp}}{\Rightarrow} \mathcal{P}_\forall(\text{Reg})$;

2. $\mathcal{P}_\exists(\text{Reg}) \overset{\text{exp}}{\Rightarrow} \text{EDTD}$;

3. $\mathcal{P}_\exists(\text{Reg}) \overset{\text{exp}}{\Rightarrow} \text{EDTD}^{\text{st}}$;

4. SIMPLIFICATION for $\mathcal{P}_\exists(\text{Reg})$ is EXPTIME-complete;

5. $\mathcal{P}_\exists(\text{Reg}) \overset{\text{exp}}{\not\to} \text{DTD}$;

6. $\mathcal{P}_\forall(\text{Reg}) \overset{\text{2-exp}}{\Rightarrow} \mathcal{P}_\exists(\text{Reg})$;

7. $\mathcal{P}_\forall(\text{Reg}) \overset{\text{2-exp}}{\Rightarrow} \text{EDTD}$;

8. $\mathcal{P}_\forall(\text{Reg}) \overset{\text{2-exp}}{\Rightarrow} \text{EDTD}^{\text{st}}$;

9. SIMPLIFICATION for $\mathcal{P}_\forall(\text{Reg})$ is EXPTIME-complete; and,

10. $\mathcal{P}_\forall(\text{Reg}) \overset{\text{2-exp}}{\not\Rightarrow} \text{DTD}$.

Proof. (1) We first show $\mathcal{P}_\exists(\text{Reg}) \overset{\text{2-exp}}{\rightarrow} \mathcal{P}_\forall(\text{Reg})$. Let $P = \{(r_1, s_1), \ldots, (r_n, s_n)\}$. We show that we can construct a complete and disjoint pattern-based schema P' such that $\mathcal{T}_\exists(P) = \mathcal{T}_\exists(P')$ in time double exponential in the size of P. By Lemma 76, $\mathcal{T}_\exists(P') = \mathcal{T}_\forall(P')$ and thus $\mathcal{T}_\exists(P) = \mathcal{T}_\forall(P')$.

For any non-empty set $C \subseteq \{1, \ldots, n\}$, denote by r_C the regular expression which defines the language $\bigcap_{i \in C} L(r_i) \setminus \bigcup_{1 \leq i \leq n, i \notin C} L(r_i)$ and by $r_\emptyset$ the expression defining $\Sigma^* \setminus \bigcup_{1 \leq i \leq n} L(r_i)$. That is, r_C defines any word w which is defined by all vertical expressions contained in C but is not defined by any vertical expression not contained in C. Denote by s_C the expression defining the language $\bigcup_{i \in C} L(s_i)$. Then, $P' = \{(r_\emptyset, \emptyset)\} \cup \{(r_C, s_C) \mid C \subseteq \{1, \ldots, n\} \wedge C \neq \emptyset\}$. Here, P' is disjoint and complete. We show that $\mathcal{T}_\exists(P) = \mathcal{T}_\exists(P')$. By Lemma 75(3), it suffices to prove that for any node v of any tree t, $v \in P_\exists(t)$ if and only if $v \in P'_\exists(t)$:

- $v \in P_\exists(t) \Rightarrow v \in P'_\exists(t)$: Let $C = \{i \mid \text{anc-str}(v) \in L(r_i)\}$. Since $v \in P_\exists(t)$, $C \neq \emptyset$ and there is an $i \in C$ with $\text{ch-str}(v) \in L(s_i)$. But then, by definition of r_C and s_C, $\text{anc-str}(v) \in L(r_C)$ and $\text{ch-str}(v) \in L(s_C)$, and thus $v \in P'_\exists(t)$.

- $v \in P'_\exists(t) \Rightarrow v \in P_\exists(t)$: Let $C \subseteq \{1, \ldots, n\}$ be the unique set for which $\text{anc-str}(v) \in L(r_C)$ and $\text{ch-str}(v) \in L(s_C)$, and choose some $i \in C$ for which $\text{ch-str}(v) \in L(s_i)$. By definition of s_C, such an i must exist. Then, $\text{anc-str}(v) \in L(r_i)$ and $\text{ch-str}(v) \in L(s_i)$, from which it follows that $v \in P_\exists(t)$.

We conclude by showing that P' can be constructed from P in time double exponential in the size of P. By Theorem 82(1), the expressions r_C can be constructed in time double exponential in the size of the r_i and s_i. The expressions s_C can easily be constructed in linear time by taking the disjunction of the right expressions. So, any rule (r_C, s_C) requires at most double exponential time to construct, and we must construct an exponential number of these rules, which yields and algorithm of double exponential time complexity.

To show that $\mathcal{P}_\exists(\text{Reg}) \overset{\text{2-exp}}{\Rightarrow} \mathcal{P}_\forall(\text{Reg})$, we slightly extend Theorem 15.

Lemma 84. For every $n \in \mathbb{N}$, there is a regular expressions r_n of size linear in n such that any regular expression r defining $\Sigma^* \setminus L(r_n)$ is of size at least double exponential in r. Further, r_n has the property that for any string $w \notin L(r_n)$, there exists a string u such that $wu \in L(r_n)$.

Proof. Let $n \in \mathbb{N}$. By Theorem 15, there exists a regular expression s_n of size linear in n over an alphabet Σ such that any regular expression defining $\Sigma^* \setminus L(s_n)$ must be of size at least double exponential in n. Let $\Sigma_a = \Sigma \uplus \{a\}$. Define $r_n = s_n + \Sigma_a^* a$ as all strings which are defined by s_n or have a as last symbol. First, note that r_n satisfies the extra condition: for every $w \notin L(r_n)$, $wa \in L(r_n)$. We show that any expression r defining the complement of r_n must be of size at least double exponential in n. This complement consists of all strings which do not have a as last symbol and are not defined by s_n. But then, the expression s which defines $L(r) \cap \Sigma^*$ defines exactly $L(s_n) \setminus \Sigma^*$, the complement of $L(s_n)$. Furthermore, by Theorem 15, s must be of size at least double exponential in n and by Theorem 82(2), s can be computed from r in time linear in the size of r. It follows that r must also be of size at least double exponential in n. □

Now, let $n \in \mathbb{N}$ and let r_n be a regular expression over Σ satisfying the conditions of Lemma 84. Then, define $P_n = \{(r_n, \varepsilon), (\Sigma^*, \Sigma)\}$. Here, $\mathcal{T}_\exists(P_n)$ defines all unary trees w for which $w \in L(r_n)$.

Let P be a pattern-based schema with $\mathcal{T}_\exists(P_n) = \mathcal{T}_\forall(P)$. Define $U = \{r \mid (r, s) \in P \wedge \varepsilon \notin L(s)\}$ as the set of vertical regular expressions in P whose corresponding horizontal regular expression does not contain the empty string. Finally, let r be the disjunction of all expressions in U. We now show that $L(r) = \Sigma^* \setminus L(r_n)$, thereby proving that the size of P must be at least double exponential in n.

First, let $w \notin L(r_n)$ and towards a contradiction suppose $w \notin L(r)$. Then, $w \notin \mathcal{T}_\exists(P_n) = \mathcal{T}_\forall(P)$. By Lemma 84, there exists a string u such that $wu \in L(r_n)$, and thus $wu \in \mathcal{T}_\exists(P_n)$ by definition of P_n and so $wu \in \mathcal{T}_\forall(P)$. By Lemma 75(2), for every non-leaf node v of w, $v \in P_\forall(w)$. As w is not defined by any expression in U, for any rule $(r', s') \in P$ with $w \in L(r')$ it holds that $\varepsilon \in L(s')$, and thus for the leaf node v of w, $v \in P_\forall(w)$. So, by Lemma 75(1), $w \in \mathcal{T}_\forall(P)$ which leads to the desired contradiction.

Conversely, suppose $w \in L(r')$, for some $r' \in U$, and again towards a contradiction suppose $w \in L(r_n)$. Then, $w \in \mathcal{T}_\exists(P) = \mathcal{T}_\forall(P)$. But, since

$w \in L(r')$, and by definition of U for the rule (r', s') in P it holds that $\varepsilon \notin L(s')$. It follows that the leaf node v of w is not in $P_\forall(w)$. Therefore, $w \notin \mathcal{T}_\forall(P)$ by Lemma 75(1), which again gives us the desired contradiction. This concludes the proof of Theorem 83(1).

(2-3) We first show $\mathcal{P}_\exists(\text{Reg}) \overset{\text{exp}}{\rightarrow} \text{EDTD}^{\text{st}}$, which implies $\mathcal{P}_\exists(\text{Reg}) \overset{\text{exp}}{\rightarrow} \text{EDTD}$.

Thereto, let $P = \{(r_1, s_1), \dots, (r_n, s_n)\}$. We construct an automaton-based schema $D = (A, \lambda)$ such that $L(D) = \mathcal{T}_\exists(P)$. By Lemma 78, D can then be translated into an equivalent single-type EDTD in polynomial time and the theorem follows. First, construct for every r_i a DFA $A_i = (Q_i, q_i, \delta_i, F_i)$, such that $L(r_i) = L(A_i)$. Then, $A = (Q_1 \times \cdots \times Q_n, (q_1, \dots, q_n), \delta)$ is the product automaton for $A_1, \dots, A_n$. Finally, $\lambda((q_1, \dots, q_n)) = \bigcup_{i \leq n, q_i \in F_i} L(s_i)$, and $\lambda((q_1, \dots, q_n)) = \emptyset$ if none of the q_i are accepting states for their automaton. Here, if m is the size of the largest vertical expression in P, then A is of size $O(2^{m \cdot n})$. Furthermore, an expression for $\bigcup_{i \leq n, q_i \in F_i} L(s_i)$ is simply the disjunction of these s_i and can be constructed in linear time. Therefore, the total construction can be carried out in exponential time.

Further, $\mathcal{P}_\exists(\text{Reg}) \overset{\text{exp}}{\Rightarrow} \text{EDTD}$ already holds for a restricted version of pattern-based schemas, which is shown in Theorem 85(2). The latter then implies $\mathcal{P}_\exists(\text{Reg}) \overset{\text{exp}}{\Rightarrow} \text{EDTD}^{\text{st}}$.

(4) For the upper bound, we combine a number of results of Kasneci and Schwentick [62] and Martens et. al [77]. In the following, an NTA(NFA) is a non-deterministic tree automaton where the transition relation is represented by an NFA. Recall also that DTD(NFA) is a DTD where content models are defined by NFAs.

Given a pattern-based schema P, we first construct an NTA(NFA) A_P with $L(A_P) = \mathcal{T}_\exists(P)$, which can be done in exponential time (Proposition 3.3 in [62]). Then, Martens et. al. [77] have shown that given any NTA(NFA) A_P it is possible to construct, in time polynomial in the size of A_P, a DTD(NFA) D_P such that $L(A_P) \subseteq L(D_P)$ is always true and $L(A_P) = L(D_P)$ holds if and only if $L(A_P)$ is definable by a DTD. Summarizing, D_P is of size exponential in P, $\mathcal{T}_\exists(P) \subseteq L(D_P)$ and $\mathcal{T}_\exists(P)$ is definable by a DTD if and only if $\mathcal{T}_\exists(P) = L(D_P)$.

Now, construct another NTA(NFA) $A_{\neg P}$ which defines the complement of $\mathcal{T}_\exists(P)$. This can again be done in exponential time (Proposition 3.3 in [62]). Since $\mathcal{T}_\exists(P) \subseteq L(D_P)$, $\mathcal{T}_\exists(P) = L(D_P)$ if and only if $L(D_P) \cap L(A_{\neg P}) \neq \emptyset$. Here, D_P and $A_{\neg P}$ are of size at most exponential in the size of P, and testing the non-emptiness of their intersection can be done in time polynomial in the size of D_P and $A_{\neg P}$. This gives us an EXPTIME algorithm overall.

For the lower bound, we reduce from SATISFIABILITY of pattern-based schemas, which is EXPTIME-complete [62]. Let P be a pattern-based schema over the alphabet Σ, define $\Sigma^P = \{a, b, c, e\} \uplus \Sigma$, and define the pattern-based schema $P' = \{(a, b + c), (ab, e), (ac, e), (abe, \varepsilon), (ace, \varepsilon)\} \cup \{(acer, s) \mid (r, s) \in P\}$. We show that $\mathcal{T}_\exists(P')$ is definable by a DTD if and only if P is not existentially satisfiable. Since EXPTIME is closed under complement, the theorem follows.

If $\mathcal{T}_\exists(P) = \emptyset$, then the following DTD d defines $\mathcal{T}_\exists(P')$: $d(a) = b + c$, $d(b) = e$, $d(c) = e$, $d(e) = \varepsilon$.

Conversely, if there exists some tree $t \in \mathcal{T}_\exists(P)$, suppose towards a contradiction that there exists a DTD D such that $L(D) = \mathcal{T}_\exists(P')$. Then, $a(b(e)) \in L(D)$, and $a(c(e(t))) \in L(D)$. Since every DTD is closed under label-guarded subtree exchange (Lemma 79), $a(b(e(t))) \in L(D)$ also holds, but $a(b(e(t))) \notin \mathcal{T}_\exists(P')$ which yields the desired contradiction.

(5) First, $\mathcal{P}_\exists(\text{Reg}) \not\rightarrow \text{DTD}$ already holds for a restricted version of pattern-based schemas (Theorem 89(6)). We show $\mathcal{P}_\exists(\text{Reg}) \overset{\text{exp}}{\not\rightarrow} \text{DTD}$.

Simply translating the DTD(NFA), obtained in the previous proof, into a normal DTD by means of state elimination would give us a double exponential algorithm. Therefore, we use the following similar approach which does not need to translate regular expressions into NFAs and back. First, construct a single-type EDTD D_1 such that $L(D_1) = \mathcal{T}_\exists(P)$. This can be done in exponential time according to Theorem 83(3). Then, use the polynomial time algorithm of Martens et al [77], to construct an equivalent DTD D. In this algorithm, all expressions of D define unions of the language defined by the expressions in D_1. This can, of course, be done by taking the disjunction of expressions in D_1. In total, D is constructed in exponential time.

(6) We first show $\mathcal{P}_\forall(\text{Reg}) \overset{\text{2-exp}}{\rightarrow} \mathcal{P}_\exists(\text{Reg})$. We take the same approach as in the proof of Theorem 83(1), but have to make some small changes. Let $P = \{(r_1, s_1), \ldots, (r_n, s_n)\}$, and for any non-empty set $C \subseteq \{1, \ldots, n\}$ let r_C be the regular expression defining $\bigcap_{i \in C} L(r_i) \setminus \bigcup_{1 \leq i \leq n, i \notin C} L(r_i)$. Let $r_\emptyset$ define $\Sigma^* \setminus \bigcap_{i \leq n} L(r_i)$ and let s_C be the expression defining the language $\bigcap_{i \in C} L(s_i)$. Define $P' = \{(r_\emptyset, \Sigma^*)\} \cup \{(r_C, s_C \mid C \subseteq \{1, \ldots, n\} \wedge C \neq \emptyset\}$. Here, P' is disjoint and complete and, by the same argumentation as in the proof of Theorem 83(1), can be constructed in time double exponential in the size of P'. So, by Lemma 76, $\mathcal{T}_\exists(P') = \mathcal{T}_\forall(P')$. We show that $\mathcal{T}_\forall(P) = \mathcal{T}_\forall(P')$ from which $\mathcal{T}_\forall(P) = \mathcal{T}_\exists(P')$ then follows. By Lemma 75(1), it suffices to prove that for any node v of any tree t, $v \in P_\forall(t)$ if and only if $v \in P'_\forall(t)$:

- $v \in P_\forall(t) \Rightarrow v \in P'_\forall(t)$: Let $C = \{i \mid \text{anc-str}(v) \in L(r_i)\}$. If $C = \emptyset$, then anc-str$(v) \in L(r_\emptyset)$ and the horizontal regular expression Σ^* allows every child-string. Because of the disjointness of P' no other vertical regular expression in P' can define anc-str(v) and thus $v \in P'_\forall(t)$. If $C \neq \emptyset$, since $v \in P_\forall(t)$, for all $i \in C$, ch-str$(v) \in L(s_i)$. But then, by definition of r_C and s_C, anc-str$(v) \in L(r_C)$ and ch-str$(v) \in L(s_C)$, combined with the disjointness of P' gives $v \in P'_\forall(t)$.

- $v \in P'_\forall(t) \Rightarrow v \in P_\forall(t)$: Let $C \subseteq \{1, \dots, n\}$ be the unique set for which $(r_C, s_C) \in P'$, anc-str$(v) \in L(r_C)$ and ch-str$(v) \in L(s_C)$. Since $v \in P'_\forall(t)$ and by the disjointness and completeness of P' there indeed exists exactly one such set. If $C = \emptyset$, then anc-str(v) is not defined by any vertical expression in P and thus $v \in P_\forall(t)$. If $C \neq \emptyset$, then for all $i \in C$, anc-str$(v) \in L(r_i)$ and ch-str$(v) \in L(s_i)$, and for all $i \notin C$, anc-str$(v) \notin L(r_i)$. It follows that $v \in P_\forall(t)$.

We now show that $\mathcal{P}_\forall(\text{Reg}) \overset{\text{2-exp}}{\Rightarrow} \mathcal{P}_\exists(\text{Reg})$. Let $n \in \mathbb{N}$. According to Theorem 20(3), there exist a linear number of regular expressions $r_1, \dots, r_m$ of size linear in n such that any regular expression defining $\bigcap_{i \leq m} L(r_i)$ must be of size at least double exponential in n. For brevity, define $K = \bigcap_{i \leq m} L(r_i)$.

Define P_n over the alphabet $\Sigma_a = \Sigma \uplus \{a\}$, for $a \notin \Sigma$, as $P_n = \{(a, r_i) \mid i \leq m\} \cup \{(ab, \varepsilon) \mid b \in \Sigma\} \cup \{(b, \emptyset) \mid b \in \Sigma\}$. That is, $\mathcal{T}_\forall(P_n)$ contains all trees $a(w)$, where $w \in K$.

Let P be a pattern-based schema with $\mathcal{T}_\forall(P_n) = \mathcal{T}_\exists(P)$. For an expression s, denote by s^- the expression defining all words in $L(s) \cap \Sigma^*$. According to Theorem 82(2), s^- can be constructed from s in linear time. Define $U = \{s^- \mid (r, s) \in P \wedge a \in L(r)\}$ as the set of horizontal regular expressions whose corresponding vertical regular expressions contains the string a. Finally, let r_K be the disjunction of all expressions in U. We now show that $L(r_K) = K$, thereby proving that the size of P must be at least double exponential in n.

First, let $w \in K$. Then, $t = a(w) \in \mathcal{T}_\forall(P_n) = \mathcal{T}_\exists(P)$. Therefore, by Lemma 75(3), the root node v of t is in $P_\exists(t)$. It follows that there must be a rule $(r, s) \in P$, with $a \in L(r)$ and $w \in L(s)$. Now $w \in \Sigma^*$ implies $w \in L(s^-)$, and thus, by definition of U and r_K, $w \in L(r_K)$.

Conversely, suppose $w \in L(s^-)$ for some $s^- \in U$. We show that $t = a(w) \in \mathcal{T}_\exists(P) = \mathcal{T}_\forall(P_n)$, which implies that $w \in K$. By Lemma 75(3), it suffices to show that every node v of t is in $P_\exists(t)$. For the root node v of t, we know that ch-str$(v) = w \in L(s^-)$, and by definition of U, that anc-str$(v) = a \in L(r)$, where r is the corresponding vertical expression for s. Therefore, $v \in P_\exists(t)$. All other nodes v are leaf nodes with ch-str$(v) = \varepsilon$ and anc-str$(v) = ab$, where $b \in \Sigma$ since $w \in L(s^-)$. To show that any node with these child and ancestor-strings must be in $P_\exists(t)$, note that for every symbol $b \in \Sigma$ there exists a string $w' \in K$ such that w' contains a b. Otherwise b is useless and can be removed from Σ. Then, $t' = a(w') \in \mathcal{T}_\forall(P_n) = \mathcal{T}_\exists(P)$ and thus there is a leaf node v' in t' for which anc-str$(v') = ab$ and ch-str$(v') = \varepsilon$. Since, by Lemma 75(3)

$v' \in P_\exists(t')$, also any leaf node v of t with anc-str$(v) = ab$ is in $P_\exists(t)$. It follows that $t \in \mathcal{T}_\exists(P) = \mathcal{T}_\forall(P_n)$.

(7-8) We first show $\mathcal{P}_\forall(\text{Reg}) \overset{\text{2-exp}}{\rightarrow} \text{EDTD}^{\text{st}}$, which implies $\mathcal{P}_\forall(\text{Reg}) \overset{\text{2-exp}}{\rightarrow} \text{EDTD}$. Thereto, let $P = \{(r_1, s_1), \ldots, (r_n, s_n)\}$. We construct an automaton-based schema $D = (A, \lambda)$ such that $L(D) = \mathcal{T}_\forall(P)$. By Lemma 78, D can then be translated into an equivalent single-type EDTD and the theorem follows. We construct A in exactly the same manner as in the proof of Theorem 83(3). For λ, let $\lambda((q_1, \ldots, q_n)) = \bigcap_{i \leq n, q_i \in F_i} L(s_i)$, and $\lambda((q_1, \ldots, q_n)) = \Sigma^*$ if none of the q_i are accepting states for their automaton. We already know that A can be constructed in exponential time, and by Theorem 20(1) a regular expression for $\lambda((q_1, \ldots, q_n)) = \bigcap_{i \leq n, q_i \in F_i} L(s_i)$ can be constructed in double exponential time. It follows that the total construction can be done in double exponential time.

Further, $\mathcal{P}_\forall(\text{Reg}) \overset{\text{2-exp}}{\Rightarrow} \text{EDTD}$ already holds for a restricted version of pattern-based schemas, which is shown in Theorem 85(7). The latter implies $\mathcal{P}_\forall(\text{Reg}) \overset{\text{2-exp}}{\Rightarrow} \text{EDTD}^{\text{st}}$.

(9) The proof is along the same lines as that of Theorem 83(4).

(10) First, $\mathcal{P}_\forall(\text{Reg}) \not\rightarrow \text{DTD}$ already holds for a restricted version of pattern-based schemas (Theorem 89(12)).

We first show $\mathcal{P}_\forall(\text{Reg}) \overset{\text{2-exp}}{\not\rightarrow} \text{DTD}$. Notice that the DTD(NFA) D constructed in the above proof, conform the proof of Theorem 83(4), is constructed in time exponential in the size of P. To obtain an actual DTD, we only have to translate the NFAs in D into regular expressions, which can be done in exponential time (Theorem 2(1)). This yields a total algorithm of double exponential time complexity.

Finally, $\mathcal{P}_\forall(\text{Reg}) \overset{\text{2-exp}}{\not\Rightarrow} \text{DTD}$ already holds for a more restricted version of pattern-based schemas, which is shown in Theorem 85(10). □

7.3 Linear Pattern-Based Schemas

In this section, following [62], we restrict the vertical expressions to XPath expressions using only descendant and child axes. For instance, an XPath expression $\backslash\backslash a \backslash\backslash b \backslash c$ captures all nodes that are labeled with c, have b as parent and have an a as ancestor. This corresponds to the regular expression $\Sigma^* a \Sigma^* bc$.

Formally, we call an expression *linear* if it is of the form $w_0\Sigma^* \cdots w_{n-1}\Sigma^* w_n$, with $w_0, w_n \in \Sigma^*$, and $w_i \in \Sigma^+$ for $1 \leq i < n$. A pattern-based schema is *linear* if all its vertical expressions are linear. Denote the classes of linear schemas under existential and universal semantics by $\mathcal{P}_\exists(\mathrm{Lin})$ and $\mathcal{P}_\forall(\mathrm{Lin})$, respectively.

Theorem 85 lists the results for linear schemas. The complexity of SIMPLIFICATION improves slightly, PSPACE instead of EXPTIME. Further, we show that the expressive power of linear schemas under existential and universal semantics becomes incomparable, but that the complexity of translating to DTDs and (single-type) EDTDs is in general not better than for regular pattern-based schemas.

Theorem 85.

1. $\mathcal{P}_\exists(\mathrm{Lin}) \not\rightarrow \mathcal{P}_\forall(\mathrm{Lin})$;
2. $\mathcal{P}_\exists(\mathrm{Lin}) \overset{\mathrm{exp}}{\Rightarrow} \mathrm{EDTD}$;
3. $\mathcal{P}_\exists(\mathrm{Lin}) \overset{\mathrm{exp}}{\Rightarrow} \mathrm{EDTD}^{\mathrm{st}}$;
4. SIMPLIFICATION for $\mathcal{P}_\exists(\mathrm{Lin})$ is PSPACE-complete;
5. $\mathcal{P}_\exists(\mathrm{Lin}) \overset{\mathrm{exp}}{\not\rightarrow} \mathrm{DTD}$;
6. $\mathcal{P}_\forall(\mathrm{Lin}) \not\rightarrow \mathcal{P}_\exists(\mathrm{Lin})$;
7. $\mathcal{P}_\forall(\mathrm{Lin}) \overset{\text{2-exp}}{\Rightarrow} \mathrm{EDTD}$;
8. $\mathcal{P}_\forall(\mathrm{Lin}) \overset{\text{2-exp}}{\Rightarrow} \mathrm{EDTD}^{\mathrm{st}}$;
9. SIMPLIFICATION for $\mathcal{P}_\forall(\mathrm{Lin})$ is PSPACE-complete; and,
10. $\mathcal{P}_\forall(\mathrm{Lin}) \overset{\text{2-exp}}{\not\Rightarrow} \mathrm{DTD}$.

Proof. (1) We first prove the following simple lemma. Given an alphabet Σ, and a symbol $b \in \Sigma$, denote $\Sigma \setminus \{b\}$ by Σ_b.

Lemma 86. There does not exist a set of linear regular expression $r_1, \dots, r_n$ such that $\bigcup_{1\le i\le n} L(r_i)$ is an infinite language and $\bigcup_{1\le i\le n} L(r_i) \subseteq L(\Sigma_b^*)$.

Proof. Suppose to the contrary that such a list of linear expressions does exist. Then, one of these expressions must contain Σ^* because otherwise $\bigcup_{1\le i\le n} L(r_i)$ would be a finite language. However, if an expression contains Σ^*, then it also defines words containing b, which gives us the desired contradiction. □

Now, let $P = \{(\Sigma^* b \Sigma^*, \varepsilon), (\Sigma^*, \Sigma)\}$. Then, $\mathcal{T}_\exists(P)$ defines all unary trees containing at least one b. Suppose that P' is a linear schema such that $\mathcal{T}_\exists(P) = \mathcal{T}_\forall(P')$. Define $U = \{r \mid (r, s) \in P'$ and $\varepsilon \notin L(s)\}$ as the set of all vertical regular expressions in P' whose horizontal regular expressions do not contain the empty string. We show that the union of the expressions in U defines an infinite language and is a subset of Σ_b^*, which by Lemma 86 proves that such a schema P' can not exist.

First, to show that the union of these expressions defines an infinite language, suppose that it does not. Then, every expression $r \in U$ is of the form $r = w$, for some string w. Let k be the length of the longest such string w. Now, $a^{k+1}b \in \mathcal{T}_\exists(P) = \mathcal{T}_\forall(P')$ and thus by Lemma 75(2) every non-leaf node v of a^{k+1} is in $P'_\forall(a^{k+1})$. Further, $a^{k+1} \notin L(r)$ for all vertical expressions in U and thus the leaf node of a^{k+1} is also in $P'_\forall(a^{k+1})$. But then, by Lemma 75(1), $a^{k+1} \in \mathcal{T}_\forall(P')$ which leads to the desired contradiction.

Second, let $w \in L(r)$, for some $r \in U$, we show $w \in \Sigma_b^*$. Towards a contradiction, suppose $w \notin \Sigma_b^*$, which means that w contains at least one b and thus $w \in \mathcal{T}_\exists(P) = \mathcal{T}_\forall(P')$. But then, for the leaf node v of w, anc-str$(v) = w \in L(r)$, and by definition of U, ch-str$(v) = \varepsilon \notin L(s)$, where s is the corresponding horizontal expression for r. Then, $v \notin P'_\forall(w)$ and thus by Lemma 75(1), $w \notin \mathcal{T}_\forall(P')$, which again gives the desired contradiction.

(2-3) First, $\mathcal{P}_\exists(\text{Lin}) \overset{\text{exp}}{\rightarrow} \text{EDTD}^{\text{st}}$ follows immediately from Theorem 83(3). We show $\mathcal{P}_\exists(\text{Lin}) \overset{\text{exp}}{\Rightarrow} \text{EDTD}$, which then implies both statements. Thereto, we first characterize the expressive power of EDTDs over unary tree languages.

Lemma 87. For any EDTD D for which $L(D)$ is a unary tree language, there exists an NFA A such that $L(D) = L(A)$. Moreover, A can be computed from D in time linear in the size of D.

Proof. Let $D = (\Sigma, \Sigma', d, s, \mu)$ be an EDTD, such that $L(D)$ is a unary tree language. Then, define $A = (Q, q_0, \delta, F)$ as $Q = \{q_0\} \cup \Sigma'$, $\delta = \{(q_0, s, s)\} \cup \{(a, \mu(b), b) \mid a, b \in \Sigma' \wedge b \in L(d(a))\}$, and $F = \{a \mid a \in \Sigma' \wedge \varepsilon \in d(a)\}$. □

Now, let $n \in \mathbb{N}$. Define $\Sigma_n = \{\$, \#_1, \#_2\} \cup \bigcup_{1 \leq i \leq n}\{a_i^0, a_i^1, b_i^0, b_i^1\}$ and $K_n = \{\#_1 a_1^{i_1} a_2^{i_2} \cdots a_n^{i_n} \$ b_1^{i_1} b_2^{i_2} \cdots b_n^{i_n} \#_2 \mid i_k \in \{0,1\}, 1 \leq k \leq n\}$. It is not hard to see that any NFA defining K_n must be of size at least exponential in n. Indeed define $M = \{(x, w) \mid xw \in K_n \wedge |x| = n+1\}$ which is of size exponential in n, and satisfies the conditions of Theorem 81. Then, by Lemma 87, every EDTD defining the unary tree language K_n must also be of size exponential in n. We conclude the proof by giving a pattern-based schema P_n, such that $\mathcal{T}_\exists(P_n) = K_n$, which is of size linear in n. It contains the following rules:

- $\#_1 \rightarrow a_1^0 + a_1^1$
- For any $i < n$:
  - $\#_1 \Sigma^* a_i^0 \rightarrow a_{i+1}^0 + a_{i+1}^1$

  - $\#_1\Sigma^* a_i^1 \to a_{i+1}^0 + a_{i+1}^1$
  - $\#_1\Sigma^* a_i^0 \Sigma^* b_i^0 \to b_{i+1}^0 + b_{i+1}^1$
  - $\#_1\Sigma^* a_i^1 \Sigma^* b_i^1 \to b_{i+1}^0 + b_{i+1}^1$

- $\#_1\Sigma^* a_n^0 \to \$$
- $\#_1\Sigma^* a_n^1 \to \$$
- $\#_1\Sigma^* \$ \to b_1^0 + b_1^1$
- $\#_1\Sigma^* a_n^0 \Sigma^* b_n^0 \to \#_2$
- $\#_1\Sigma^* a_n^1 \Sigma^* b_n^1 \to \#_2$
- $\#_1\Sigma^* \#_2 \to \varepsilon$

(4) For the lower bound, we reduce from UNIVERSALITY of regular expressions. That is, deciding for a regular expression r whether $L(r) = \Sigma^*$. The latter problem is known to be PSPACE-complete [102]. Given r over alphabet Σ, let $\Sigma^P = \{a, b, c, d\} \uplus \Sigma$, and define the pattern-based schema $P = \{(a, b + c), (ab, e), (ac, e), (abe, \Sigma^*), (ace, r)\} \cup \{(abe\sigma, \varepsilon), (ace\sigma, \varepsilon) \mid \sigma \in \Sigma\}$. We show that there exists a DTD D with $L(D) = \mathcal{T}_\exists(P)$ if and only if $L(r) = \Sigma^*$.

If $L(r) = \Sigma^*$, then the following DTD d defines $\mathcal{T}_\exists(P)$: $d(a) = b + c$, $d(b) = e$, $d(c) = e$, $d(e) = \Sigma^*$, and $d(\sigma) = \varepsilon$ for every $\sigma \in \Sigma$.

Conversely, if $L(r) \subsetneq \Sigma^*$, we show that $\mathcal{T}_\exists(P)$ is not closed under label-guarded subtree exchange. From Lemma 79, it then follows that $\mathcal{T}_\exists(P)$ is not definable by a DTD. Let w, w' be strings such that $w \notin L(r)$ and $w' \in L(r)$. Then, $a(b(e(w))) \in L(D)$, and $a(c(e(w'))) \in L(D)$ but $a(c(e(w))) \notin \mathcal{T}_\exists(P)$.

For the upper bound, we again make use of the closure under label-guarded subtree exchange property of DTDs. Observe that $\mathcal{T}_\exists(P)$, which is a regular tree language, is not definable by any DTD if and only if there exist trees $t_1, t_2 \in \mathcal{T}_\exists(P)$ and nodes v_1 and v_2 in t_1 and t_2, respectively, with $\text{lab}^{t_1}(v_1) = \text{lab}^{t_2}(v_2)$, such that the tree $t_3 = t_1[v_1 \leftarrow \text{subtree}^{t_2}(v_2)]$ is not in $\mathcal{T}_\exists(P)$. We refer to such a tuple (t_1, t_2) as a witness to the DTD-undefinability of $\mathcal{T}_\exists(P)$, or simply a *witness tuple.*

Lemma 88. If there exists a witness tuple (t_1, t_2) for a linear schema P, then there also exists a witness tuple (t_1', t_2') for P, where t_1' and t_2' are of depth polynomial in the size of P.

Proof. We make use of techniques introduced by Kasneci and Schwentick [62]. When P, P' are two linear schemas, they stated that if there exists a tree t with $t \in \mathcal{T}_\exists(P)$ but $t \notin \mathcal{T}_\exists(P')$, then there exists a tree t' of depth polynomial with the same properties. In particular, they obtained the following property.

Let P be a linear pattern-based schema and t a tree. Then, to every node v of t, a vector $F_P^t(v)$ over $\mathbb{N}$ can be assigned with the following properties:

- along a path in a tree, $F_P^t(v)$ can take at most polynomially many values in the size of P;
- if v' is a child of v, then $F_P^t(v')$ can be computed from $F_P^t(v)$ and the label of v' in t; and
- $v \in P_\exists(t)$ can be decided solely on the value of $F_P^t(v)$ and ch-str(v).

Based on these properties it is easy to see that if there exists a tree t which existentially satisfies P, then there exists a tree t' of polynomial depth which existentially satisfies P. Indeed, t' can be constructed from t by searching for nodes v and v' of t such that v' is a descendant of v, $\text{lab}^t(v) = \text{lab}^t(v')$ and $F_P^t(v) = F_P^t(v')$, and replacing the subtree rooted at v by the one rooted at v'. By applying this rule as often as possible, we get a tree which is still existentially valid with respect to P and where no two nodes on a path in the tree have the same vector and label and which thus is of polynomial depth.

We will also use this technique, but have to be a bit more careful in the replacements we carry out. Thereto, let (t_1, t_2) be a witness tuple for P and fix nodes v_1 and v_2 of t_1 and t_2, respectively, such that t_3, defined as $t_1[v_1 \leftarrow \text{subtree}^{t_2}(v_2)]$, is not in $\mathcal{T}_\exists(P)$. Since $t_3 \notin \mathcal{T}_\exists(P)$, by Lemma 75(3), there must be some node v_3 of t_3 with $v_3 \notin P_\exists(t_3)$. Furthermore, v_3 must occur in the subtree under v_2 inherited from t_2. Indeed, every node v not in that subtree, has the same vector and child-string as its corresponding node in t_1, and since $t_1 \in \mathcal{T}_\exists(P)$ also $v \in P_\exists(t_1)$ and thus $v \in P_\exists(t_3)$. So, fix some node v_3, with $v_3 \notin P_\exists(t_3)$, occurring in t_2. Then, we can partition the trees t_1 and t_2, and thereby also t_3, in five different parts as follows:

1. $t_1[v_1 \leftarrow ()]$: the tree t_1 without the subtree under v_1;
2. $\text{subtree}^{t_1}(v_1)$: the subtree under v_1 in t_1;
3. $t_2[v_2 \leftarrow ()]$: the tree t_2 without the subtree under v_2
4. $\text{subtree}^{t_2}(v_2)[v_3 \leftarrow ()]$: the subtree under v_2 in t_2, without the subtree under v_3;
5. $\text{subtree}^{t_2}(v_3)$: the subtree under v_3 in t_2;

This situation is graphically illustrated in Figure 7.2.

Now, let t_1' and t_2' be the trees obtained from t_1 and t_2 by repeating the following as often as possible: Search for two nodes v, v' such that v is an ancestor of v', v and v' are not equal to v_1, v_2 or v_3, v and v' occur in the same

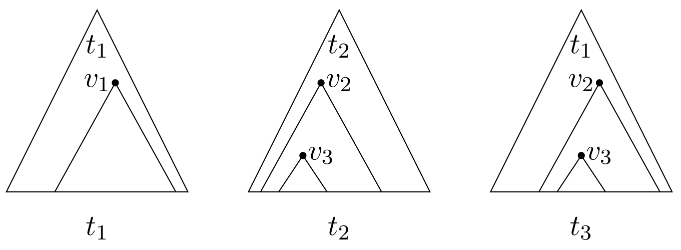


Figure 7.2: The five different areas in t_1 and t_2.

part of t_1 or t_2, $\text{lab}(v) = \text{lab}(v')$ and $F_P^{t_1}(v) = F_P^{t_1}(v')$ (or $F_P^{t_2}(v) = F_P^{t_2}(v')$ if v and v' both occur in t_2). Then, replace v by the subtree under v'.

Observe that, by the properties of F, any path in one of the five parts of t'_1 and t'_2 can have at most a polynomial depth, and thus t'_1 and t'_2 are of at most a polynomial depth. Furthermore, $t'_1, t'_2 \in \mathcal{T}_\exists(P)$ still holds and the original nodes v_1, v_2 and v_3 still occur in t'_1 and t'_2. Therefore, for $t'_3 = t'_1[v_1 \leftarrow \text{subtree}^{t'_2}(v_2)]$, $F_P^{t'_3}(v_3) = F_P^{t_3}(v_3)$ and $\text{ch-str}^{t'_3}(v_3) = \text{ch-str}^{t_3}(v_3)$. But then, $v_3 \notin P_\exists(t'_3)$, which by Lemma 75(3) gives us $t'_3 \notin \mathcal{T}_\exists(P)$. So, (t'_1, t'_2) is a witness tuple in which t'_1 and t'_2 are of at most polynomial depth. □

Now, using Lemma 88, we show that the problem is in PSPACE. We simply guess a witness tuple (t_1, t_2) and check in polynomial space whether it is a valid witness tuple. If it is, $\mathcal{T}_\exists(P)$ is not definable by a DTD. If $\mathcal{T}_\exists(P)$ is definable by a DTD, there does not exist a witness tuple for P. Since PSPACE is closed under complement, the theorem follows.

By Lemma 88, it suffices to guess trees of at most polynomial depth. Therefore, we guess t_1 and t_2 in depth-first and left-to-right fashion, maintaining for each tree and each level of the trees, the sets of states the appropriate automata can be in. Here, t_1 and t_2 are guessed simultaneously and independently. That is, for each guessed symbol, we also guess whether it belongs to t_1 or t_2. At some point in this procedure, we guess that we are now at the nodes v_1 and v_2 of t_1 and t_2. From that point we maintain a third list of states of automata, which are initiated by the values of these of t_1, but the subsequent subtree take the values of t_2. If in the end, t_1 and t_2 are accepted, but the third tree is not, then (t_1, t_2) is a valid witness for P.

(5) First, $\mathcal{P}_\exists(\text{Lin}) \not\twoheadrightarrow \text{DTD}$ already holds for a restricted version of pattern-based schemas (Theorem 89(6)). Then, $\mathcal{P}_\exists(\text{Lin}) \overset{\text{exp}}{\not\twoheadrightarrow} \text{DTD}$ follows immediately from Theorem 83(5).

(6) Let $\Sigma = \{a, b, c\}$ and define $P = \{(\Sigma^* b \Sigma^* c, b)\}$. Then, $\mathcal{T}_\forall(P)$ contains all trees in which whenever a c labeled node v has a b labeled node as ancestor,

ch-str(v) must be b. We show that any linear schema P' defining all trees in $\mathcal{T}_\forall(P)$ under existential semantics, must also define trees not in $\mathcal{T}_\forall(P)$.

Suppose there does exist a linear schema P' such that $\mathcal{T}_\forall(P) = \mathcal{T}_\exists(P')$. Define $w_\ell = a^\ell c$ for $\ell \geq 1$ and notice that $w_\ell \in \mathcal{T}_\forall(P) = \mathcal{T}_\exists(P')$. Let $(r, s) \in P'$ be a rule matching infinitely many leaf nodes of the strings w_ℓ. There must be at least one as P' contains a finite number of rules. Then, $\varepsilon \in L(s)$ must hold and r is of one of the following forms:

1. $a^{n_1}\Sigma^* a^{n_2}\Sigma^* \cdots \Sigma^* a^{n_k} c$
2. $a^{n_1}\Sigma^* a^{n_2}\Sigma^* \cdots \Sigma^* a^{n_k} c\Sigma^*$
3. $a^{n_1}\Sigma^* a^{n_2}\Sigma^* \cdots \Sigma^* a^{n_k}\Sigma^*$

where $k \geq 2$ and $n_k \geq 0$.

Choose some $N \in \mathbb{N}$ with $N \geq |P'|$ and define the unary trees $t_1 = a^N b a^N cb$ and $t_2 = a^N b a^N c$. Obviously, $t_1 \in \mathcal{T}_\forall(P)$, and $t_2 \notin \mathcal{T}_\forall(P)$. Then, $t_1 \in \mathcal{T}_\exists(P')$ and since t_2 is a prefix of t_1, by Lemma 75(4), every non-leaf node v of t_2 is in $P'_\exists(t_2)$. Finally, for the leaf node v of t_2, anc-str$(v) \in L(r)$ for any of the three expressions given above and $\varepsilon \in L(s)$ for its corresponding horizontal expression. Then, $v \in P'_\exists(t_2)$, and thus by Lemma 75(3), $t_2 \in \mathcal{T}_\exists(P')$ which completes the proof.

(7-8) First, $\mathcal{P}_\forall(\text{Lin}) \overset{\text{2-exp}}{\rightarrow} \text{EDTD}^{\text{st}}$ follows immediately from Theorem 83(3). We show $\mathcal{P}_\forall(\text{Lin}) \overset{\text{2-exp}}{\Rightarrow} \text{EDTD}$, which then implies both statements.

Let $n \in \mathbb{N}$. According to Theorem 20(3), there exist a linear number of regular expressions $r_1, \dots, r_m$ of size linear in n such that any regular expression defining $\bigcap_{i \leq m} L(r_i)$ must be of size at least double exponential in n. Set $K = \bigcap_{i \leq m} L(r_i)$.

Next, we define P_n over the alphabet $\Sigma \uplus \{a\}$ as $P_n = \{(a, r_i) \mid i \leq m\} \cup \{(ab, \varepsilon) \mid b \in \Sigma\} \cup \{(b, \emptyset) \mid b \in \Sigma\}$. That is, $\mathcal{T}_\forall(P_n)$ defines all trees $a(w)$, for which $w \in K$.

Let $D = (\Sigma, \Sigma', d, a, \mu)$ be an EDTD with $\mathcal{T}_\forall(P) = L(D)$. By Lemma 80(a), we can assume that D is trimmed. Let $a \to r$ be the single rule in D for the root element a. Let r_K be the expressions defining $\mu(L(r))$. Since D is trimmed, it follows from Lemma 80(2) that r_K cannot contain an a. But then, $L(r_K) = K$, which proves that the size of D must be at least double exponential in n.

(9) The proof is along the same lines as that of Theorem 85(4).

(10) First, $\mathcal{P}_\forall(\text{Lin}) \not\rightarrow \text{DTD}$ already holds for a restricted version of pattern-based schemas (Theorem 89(12)). Then, $\mathcal{P}_\forall(\text{Lin}) \overset{\text{2-exp}}{\not\rightarrow} \text{DTD}$ follows immediately from Theorem 83(10). For $\mathcal{P}_\forall(\text{Lin}) \overset{\text{2-exp}}{\not\Rightarrow} \text{DTD}$, let $n \in \mathbb{N}$. In the proof

of Theorem 85(7) we have defined a linear pattern-based schema P_n of size polynomial in n for which any EDTD D' with $\mathcal{T}_\forall(P_n) = L(D')$ must be of size at least double exponential in n. Furthermore, every DTD is an EDTD and the language $\mathcal{T}_\forall(P_n)$ is definable by a DTD. It follows that any DTD D with $\mathcal{T}_\forall(P_n) = L(D)$ must be of size at least double exponential in n. □

7.4 Strongly Linear Pattern-Based Schemas

In [77], it is observed that the type of a node in most real-world XSDs only depends on the labels of its parents and grand parents. To capture this idea, following [62], we say that a regular expression is *strongly linear* if it is of the form w or $\Sigma^* w$, where w is non-empty. A pattern-based schema is *strongly linear* if it is disjoint and all its vertical expressions are strongly linear. Denote the class of all strongly linear pattern-based schemas under existential and universal semantics by $\mathcal{P}_\exists$(S-Lin) and $\mathcal{P}_\forall$(S-Lin), respectively.

In [62], all horizontal expressions in a strongly linear schema are also required to be deterministic, as is the case for DTDs and XML Schema. The latter requirement is necessary to get PTIME SATISFIABILITY and INCLUSION which would otherwise be PSPACE-complete for arbitrary regular expressions. This is also the case for the SIMPLIFICATION problem studied here, but not for the various translation problems. Therefore, we distinguish between strongly linear schemas, as defined above, and strongly linear schemas where all horizontal expressions must be deterministic, which we call *deterministic strongly linear schemas* and denote by $\mathcal{P}_\exists$(Det-S-Lin) and $\mathcal{P}_\forall$(Det-S-Lin).

Theorem 89 shows the results for (deterministic) strongly linear pattern-based schemas. First, observe that the expressive power of these schemas under existential and universal semantics again coincides. Further, all considered problems become tractable, which makes strongly linear schemas very interesting from a practical point of view.

Theorem 89.
1. $\mathcal{P}_\exists$(S-Lin) $\overset{\text{poly}}{\rightarrow}$ $\mathcal{P}_\forall$(S-Lin) and $\mathcal{P}_\exists$(Det-S-Lin) $\overset{\text{poly}}{\rightarrow}$ $\mathcal{P}_\forall$(Det-S-Lin);
2. $\mathcal{P}_\exists$(S-Lin) $\overset{\text{poly}}{\rightarrow}$ EDTD and $\mathcal{P}_\exists$(Det-S-Lin) $\overset{\text{poly}}{\rightarrow}$ EDTD;
3. $\mathcal{P}_\exists$(S-Lin) $\overset{\text{poly}}{\rightarrow}$ EDTD$^{\text{st}}$ and $\mathcal{P}_\exists$(Det-S-Lin) $\overset{\text{poly}}{\rightarrow}$ EDTD$^{\text{st}}$;
4. SIMPLIFICATION for $\mathcal{P}_\exists$(S-Lin) is PSPACE-complete;
5. SIMPLIFICATION for $\mathcal{P}_\exists$(Det-S-Lin) is in PTIME;
6. $\mathcal{P}_\exists$(S-Lin) $\overset{\text{poly}}{\not\rightarrow}$ DTD and $\mathcal{P}_\exists$(Det-S-Lin) $\overset{\text{poly}}{\not\rightarrow}$ DTD;

7. $\mathcal{P}_\forall$(S-Lin) $\overset{\text{poly}}{\rightarrow}$ $\mathcal{P}_\exists$(S-Lin) and $\mathcal{P}_\forall$(Det-S-Lin) $\overset{\text{poly}}{\rightarrow}$ $\mathcal{P}_\exists$(Det-S-Lin);
8. $\mathcal{P}_\forall$(S-Lin) $\overset{\text{poly}}{\rightarrow}$ EDTD and $\mathcal{P}_\forall$(Det-S-Lin) $\overset{\text{poly}}{\rightarrow}$ EDTD;
9. $\mathcal{P}_\forall$(S-Lin) $\overset{\text{poly}}{\rightarrow}$ EDTD$^{\text{st}}$ and $\mathcal{P}_\forall$(Det-S-Lin) $\overset{\text{poly}}{\rightarrow}$ EDTD$^{\text{st}}$;
10. SIMPLIFICATION for $\mathcal{P}_\forall$(S-Lin) is PSPACE-complete;
11. SIMPLIFICATION for $\mathcal{P}_\forall$(Det-S-Lin) is in PTIME; and,
12. $\mathcal{P}_\forall$(S-Lin) $\overset{\text{poly}}{\not\rightarrow}$ DTD and $\mathcal{P}_\forall$(Det-S-Lin) $\overset{\text{poly}}{\not\rightarrow}$ DTD.

Proof. (1) We first show $\mathcal{P}_\exists$(S-Lin) $\overset{\text{poly}}{\rightarrow}$ $\mathcal{P}_\forall$(S-Lin). The key of this proof lies in the following lemma:

Lemma 90. For each finite set R of disjoint strongly linear expressions, a finite set S of disjoint strongly linear regular expressions can be constructed in polynomial time such that $\bigcup_{s\in S} L(s) = \Sigma^* \setminus \bigcup_{r\in R} L(r)$.

Before we prove this lemma, we show how it implies the theorem. For $P = \{(r_1, s_1), \dots, (r_n, s_n)\}$, let S be the set of strongly linear expressions for $R = \{r_1, \dots, r_n\}$ satisfying the conditions of Lemma 90. Set $P' = P \cup \bigcup_{s\in S}\{(s, \emptyset)\}$. Here, $\mathcal{T}_\exists(P) = \mathcal{T}_\exists(P')$ and since P' is disjoint and complete it follows from Lemma 76 that $\mathcal{T}_\exists(P') = \mathcal{T}_\forall(P')$. This gives us $\mathcal{T}_\exists(P) = \mathcal{T}_\forall(P')$. By Lemma 90, the set S is polynomial time computable and therefore, P' is too.

Further, note that the regular expressions in P' are copies of these in P. Therefore, $\mathcal{P}_\forall$(Det-S-Lin) $\overset{\text{poly}}{\rightarrow}$ $\mathcal{P}_\exists$(Det-S-Lin) also holds. We finally give the proof of Lemma 90.

Proof. For R a set of strongly linear regular expressions, let $\text{Suffix}(R) = \bigcup_{r\in R}\text{Suffix}(r)$. Define U as the set of strings aw, $a \in \Sigma$, $w \in \Sigma^*$, such that $w \in \text{Suffix}(R)$, and $aw \notin \text{Suffix}(R)$. Define V as $\text{Suffix}(R) \setminus \bigcup_{r\in R} L(r)$.

We claim that $S = \bigcup_{u\in U}\{\Sigma^* u\} \cup \bigcup_{v\in V}\{v\}$ is the desired set of regular expressions. For instance, for $R = \{\Sigma^* abc, \Sigma^* b, bc\}$ we have $U = \{bbc, cbc, ac, cc, a\}$ and $V = \{c\}$ which gives us $S = \{\Sigma^* bbc, \Sigma^* cbc, \Sigma^* ac, \Sigma^* cc, \Sigma^* a, c\}$.

It suffices to show that, given R: (1) S is finite and polynomial time computable; (2) the expressions in S are pairwise disjoint; (3) $\bigcup_{r\in R} L(r) \cap \bigcup_{s\in S} L(s) = \emptyset$; and, (4) $\bigcup_{r\in R\cup S} L(r) = \Sigma^*$.

We first show (1). Every $r \in R$ is of the form w or $\Sigma^* w$, for some w. Then, for r there are only $|w|$ suffixes in $L(r)$ which can match the definition of U or V. When a string w', with $|w'| > |w|$, is a suffix in $L(r)$ then r must be of the form $\Sigma^* w$ and thus for every $a \in \Sigma$, aw is also a suffix in $L(r)$, and thus

$aw \notin U$. Further, $w' \notin V$. So, the number of strings in U and V is bounded by the number of rules in R times the length of the strings w occurring in the expressions in R, times the number of alphabet symbols, which is a polynomial. Obviously, we can also compute these strings in polynomial time.

For (2), we must check that the generated expressions are all pairwise disjoint. First, every expression generated by V defines only one string, so two expressions generated by V always have an empty intersection. For an expression $\Sigma^* aw$ generated by U and an string w' in V, suppose that their intersection is non-empty and thus $w' \in L(\Sigma^* aw)$. Then, aw must be a suffix of w' and we know by definition of V that $w' \in \text{Suffix}(R)$. But then, also $aw \in \text{Suffix}(R)$ which contradicts the definition of U. Third, suppose that two expressions $\Sigma^* aw, \Sigma^* a'w'$ generated by U have a non-empty intersection. Then, aw must be a suffix of $a'w'$ (or the other way around, but that is perfectly symmetrical), and since $aw \neq a'w'$, aw must be a suffix of w'. But $w' \in \text{Suffix}(R)$ and thus $aw \in \text{Suffix}(R)$ must also hold, which again contradicts the definition of U.

For (3), The strings in V are explicitly defined such that their intersection with $\bigcup_{r \in R} L(r)$ is empty. For the expression generated by U, observe that they only define words which have suffixes that can not be suffixes of any word defined by any expression in R. Therefore, $\bigcup_{r \in R} L(r) \cap \bigcup_{s \in S} L(s) = \emptyset$.

Finally, we show (4). Let $w \notin L(r)$, for any $r \in R$. We show that there exists an $s \in S$, such that $w \in L(s)$. If $w \in V$, we are done. So assume $w \notin V$. Let $w = a_1 \cdots a_k$. Now, we go from left to right through w and search for the rightmost $l \leq k+1$ such that $w_l = a_l \cdots a_k \in \text{Suffix}(R)$, and $w_{l-1} = a_{l-1} \cdots a_k \notin \text{Suffix}(R)$. When $l = k+1$, $w_l = \varepsilon$. Then, w is accepted by the expression $\Sigma^* a_{l-1} \cdots a_k$, which by definition must be generated by U. It is only left to show that there indeed exists such an index l for w. Thereto, note that if $l = k+1$, then it is easy to see that $w_l = \varepsilon$ is a suffix of every string accepted by every $r \in R$. Conversely, if $l = 1$ we show that $w_l = w$ can not be a suffix of any string defined by any $r \in R$. Suppose to the contrary that $w \in \text{Suffix}(r)$, for some $r \in R$. Let r be w_r or $\Sigma^* w_r$. If w is a suffix of w_r, then w is accepted by an expression generated by V, which case we already ruled out. If w is not a suffix of w_r, then r must be of the form $\Sigma^* w_r$ and w_r must be a suffix of w. But then, $w \in L(r)$, which also contradicts our assumptions. So, we can only conclude that $w_1 \notin \text{Suffix}(R)$. So, given that $w_{k+1} \in \text{Suffix}(R)$, and $w_1 \notin \text{Suffix}(R)$, we are guaranteed to find some l, $1 < l \leq k+1$, such that $w_l \in \text{Suffix}(R)$, and $w_{l-1} \notin \text{Suffix}(R)$. This concludes the proof of Lemma 90. □

(7) For $P = \{(r_1, s_1), \ldots, (r_n, s_n)\}$, let $S = \{r'_1, \cdots, r'_m\}$ be the set of strongly linear expressions for $R = \{r_1, \ldots, r_n\}$ satisfying the conditions of Lemma 90.

Then, define $P' = \{(r_1, s_1), \dots, (r_n, s_n), (r'_1, \Sigma^*), \dots, (r'_m, \Sigma^*)\}$. Here, $\mathcal{T}_\forall(P) = \mathcal{T}_\forall(P')$ and since P' is disjoint and complete it follows from Lemma 76 that $\mathcal{T}_\exists(P') = \mathcal{T}_\forall(P')$. This gives us $\mathcal{T}_\forall(P) = \mathcal{T}_\exists(P')$. By Lemma 90, the set S is polynomial time computable and therefore, P' is too.

Further, note that the regular expressions in P' are copies of these in P. Therefore, $\mathcal{P}_\exists$(Det-S-Lin) $\overset{\text{poly}}{\rightarrow}$ $\mathcal{P}_\forall$(Det-S-Lin) also holds.

(2-3),(8-9) We show $\mathcal{P}_\exists$(S-Lin) $\overset{\text{poly}}{\rightarrow}$ EDTD$^{\text{st}}$. Since deterministic strongly-linear schemas are a subset of strongly-linear schemas, since single-type EDTDs are a subset of EDTDs and since we can translate a strongly-linear schema with universal semantics into an equivalent one with existential semantics in polynomial time (Theorem 89(7)), all other results follow.

Given P, we construct an automaton-based schema $D = (A, \lambda)$ such that $L(D) = \mathcal{T}_\exists(P)$. By Lemma 78, we can then translate D into an equivalent single-type EDTD in polynomial time. Let $P = \{(r_1, s_1), \dots, (r_n, s_n)\}$. We define D such that when A is in state q after reading w, $\lambda(q) = s_i$ if and only if $w \in L(r_i)$ and $\lambda(q) = \emptyset$ otherwise. The most obvious way to construct A is by constructing DFAs for the vertical expressions and combining these by a product construction. However, this would induce an exponential blow-up. Instead, we construct A in polynomial time in a manner similar to the construction used in Proposition 5.2 in [62].

First, assume that every r_i is of the form $\Sigma^* w_i$. We later extend the construction to also handle vertical expressions of the form w_i. Define $S = \{w \mid w \in \text{Prefix}(w_i), 1 \leq i \leq n\}$. Then, $A = (Q, q_0, \delta)$ is defined as $Q = S \cup \{q_0\}$, and for each $a \in \Sigma$,

- $\delta(q_0, a) = a$ if $a \in S$, and $\delta(q_0, a) = q_0$ otherwise; and
- for each $w \in S$, $\delta(w, a) = w'$, where w' is the longest suffix of wa in S, and $\delta(w, a) = q_0$ if no string in S is a suffix of wa.

For the definition of λ, let $\lambda(q_0) = \emptyset$, and for all $w \in S$, $\lambda(w) = s_i$ if $w \in L(r_i)$ and $\lambda(w) = \emptyset$ if $w \notin L(r_i)$ for all $i \leq n$. Note that since the vertical expression are disjoint, λ is well-defined.

We prove the correctness of our construction using the following lemma which can easily be proved by induction on the length of u.

Lemma 91. For any string $u = a_1 \cdots a_k$,

1. if $q_0 \Rightarrow_{A,u} q_0$, then no suffix of u is in S; and
2. if $q_0 \Rightarrow_{A,u} w$, for some $w \in S$, then w is the biggest element in S which is a suffix of u.

3. $q_0 \Rightarrow_{A,u} q$, with $\lambda(q) = \emptyset$, if and only if $u \notin L(r_i)$, for any $i \leq n$; and

4. $q_0 \Rightarrow_{A,u} w$, $w \in S$, with $\lambda(w) = s_i$, if and only if $u \in L(r_i)$.

To show that $L(D) = \mathcal{T}_\exists(P)$, it suffices to prove that for any tree t, a node $v \in P_\exists(t)$ if and only if ch-str$(v) \in L(\lambda(q))$ for $q \in Q$ such that $q_0 \Rightarrow_{A,\text{anc-str}(v)} q$.

First, suppose $v \in P_\exists(t)$. Then, for some $i \leq n$, anc-str$(v) \in L(r_i)$ and ch-str$(v) \in L(s_i)$. By Lemma 91(4), and the definition of λ, $q_0 \Rightarrow_{\text{anc-str}(v)} q$, with $\lambda(q) = s_i$. But then, ch-str$(v) \in L(\lambda(q))$.

Conversely, suppose that for q such that $q_0 \Rightarrow_{A,\text{anc-str}(v)} q$, ch-str$(v) \in L(\lambda(q))$ holds. Then, by Lemma 91(4), there is some i such that anc-str$(v) \in L(r_i)$, and by the definition of λ, ch-str$(v) \in L(s_i)$. It follows that $v \in P_\exists(t)$.

We have now shown that the construction is correct when all expressions are of the form $\Sigma^* w$. We sketch the extension to the full class of strongly linear expressions. Assume w.l.o.g. that there exists some m such that for $i \leq m$, $r_i = \Sigma^* w_i$ and for $i > m$, $r_i = w_i$. Define $S = \{w \mid w \in \text{Prefix}(w_i) \wedge 1 \leq i \leq m\}$ in the same manner as above, and $S' = \{w \mid w \in \text{Prefix}(w_i) \wedge m < i \leq n\}$. Define $A = (Q, q_0', \delta)$, with $Q = \{q_0, q_0'\} \cup S \cup S'$. Note that the elements of S and S' need not be disjoint. Therefore, we denote the states corresponding to elements of S' by primes, for instance $ab \in S'$ corresponds to the state $a'b'$. Then, for any symbol $a \in \Sigma$, $\delta(q_0', a) = a'$ if $a \in S'$; $\delta(q_0', a) = a$ if $a \notin S' \wedge a \in S$; and $\delta(q_0', a) = q_0$ otherwise. For a string $w \in S'$, $\delta(w', a) = w'a'$ if $wa \in S'$, $\delta(w', a)$ is the longest suffix of wa in S if it exists and $wa \notin S'$, and $\delta(w', a) = q_0$ otherwise. The transition function for q_0 and the states introduced by S remains the same. So, we have added a subautomaton to A which starts by checking whether $w = w_i$, for some $i > m$, much like a suffix-tree, and switches to the normal operation of the original automaton if this is not possible anymore.

Finally, the definition of λ again remains the same for q_0 and the states introduced by S. Further, $\lambda(q_0') = \emptyset$, and $\lambda(w') = r_i$ if $w \in L(r_i)$ for some i, $1 \leq i \leq n$, and $\lambda(w') = \emptyset$ otherwise. The previous lemma can be extended for this extended construction and the correctness of the construction follows thereof.

(4),(10) This follows immediately from Theorem 85(4) and (9). The upper bound carries over since every strongly linear schema is also a linear schema. For the lower bound, observe that the schema used in the proofs of Theorem 85(4) and (9) is strongly linear.

(5),(11) We give the proof for the existential semantics. By Theorem 89(7) the result carries over immediately to the universal semantics.

The algorithm proceeds in a number of steps. First, construct an automaton-based schema D_1 such that $L(D_1) = \mathcal{T}_\exists(P)$. By Theorem 89(3) this can be done in polynomial time. Furthermore, the regular expressions in D_1 are copies of the horizontal expressions in P and are therefore also deterministic. Then, translate D_1 into a single-type EDTD $D_2 = (\Sigma, \Sigma', d_2, a, \mu)$, which by Lemma 78 can again be done in PTIME and also maintains the determinism of the used regular expressions. Then, we trim D_2 which can be done in polynomial time by Lemma 80(1) and also preserves the determinism of the expressions in D_2. Finally, we claim that $L(D_2) = \mathcal{T}_\exists(P)$ is definable by a DTD if and only if for every two types $a^i, a^j \in \Sigma'$ it holds that $L(\mu(d(a^i))) = L(\mu(d(a^j)))$. Since all regular expressions in D_2 are deterministic, this can be tested in polynomial time. We finally prove the above claim:

First, suppose that for every pair of types $a^i, a^j \in \Sigma'$ it holds that $\mu(d_2(a^i)) = \mu(d_2(a^j))$. Then, consider the DTD $D = (\Sigma, d, s)$, where $d(a) = \mu(d_2(a^i))$ for some $a^i \in \Sigma'$. Since all regular expression $\mu(d_2(a^i))$, with $\mu(a^i) = a$, are equivalent, it does not matter which type we choose. Now, $L(D) = L(D_2)$ which shows that $L(D_2)$ is definable by a DTD.

Conversely, suppose that there exist types $a^i, a^j \in \Sigma'$ such that $\mu(L(d(a^i))) \neq \mu(L(d(a^j)))$. We show that $L(D_2)$ is not closed under ancestor-guarded subtree exchange. From Lemma 79 it then follows that $L(D_2)$ is not definable by a DTD. Since $\mu(L(d(a^i))) \neq \mu(L(d(a^j)))$, there exists a string w such that $w \in \mu(L(d(a^i)))$ and $w \notin \mu(L(d(a^j)))$ or $w \notin \mu(L(d(a^i)))$ and $w \in \mu(L(d(a^j)))$. We consider the first case, the second is identical. Let $t_1 \in L(d_2)$ be a tree with some node v with $\text{lab}^{t_1}(v) = a^i$ and $\text{ch-str}^{t_1}(v) = w'$ where $\mu(w') = w$. Further, let $t_2 \in L(d_2)$ be a tree with some node u with $\text{lab}^{t_2}(u) = a^j$. Since D_2 is trimmed, t_1 and t_2 must exist by Lemma 80(2). Now, define $t_3 = \mu(t_2)[u \leftarrow \mu(\text{subtree}^{t_1}(v))]$ which is obtained from $\mu(t_1)$ and $\mu(t_2)$ by label-guarded subtree exchange. Because D_2 is a single-type EDTD, it must assign the type a^j to node u in t_3. However, $\text{ch-str}^{t_3}(u) = w \notin \mu(L(d(a^j)))$ and thus $t_3 \notin L(D_3)$. This shows that D_2 is not closed under label-guarded subtree exchange.

(6),(12) We first show that $\mathcal{P}_\forall(\text{Det-S-Lin}) \not\rightarrow \text{DTD}$ and then $\mathcal{P}_\exists(\text{S-Lin}) \overset{\text{poly}}{\not\rightarrow}$ DTD. Since deterministic strongly-linear schemas are a subset of strongly-linear schemas and since we can translate a strongly-linear schema with universal semantics into an equivalent one with existential semantics in polynomial time (Theorem 89(7)), all other results follow.

First, to show that $\mathcal{P}_\forall(\text{Det-S-Lin}) \not\rightarrow \text{DTD}$, let $\Sigma^P = \{a, b, c, d, e, f\}$ and $P = \{(a, b + c), (ab, d), (ac, d), (abd, \varepsilon), (acd, f), (acdf, \varepsilon)\}$. Here, $a(b(d)) \in \mathcal{T}_\forall(P)$ and $a(c(d(f))) \in \mathcal{T}_\forall(P)$ but $a(b(d(f))) \notin \mathcal{T}_\forall(P)$. Therefore, $\mathcal{T}_\forall(P)$ is not closed under ancestor-guarded subtree exchange and by Lemma 79 is not definable by a DTD.

To show that $\mathcal{P}_\exists(\text{S-Lin}) \overset{\text{poly}}{\not\rightarrow}$ DTD, note that the algorithm in the above proof also works when the horizontal regular expressions are not deterministic. The total algorithm then becomes PSPACE, because we have to test equivalence of regular expressions. However, the DTD D is still constructed in polynomial time, which completes this proof. $\square$

8 Optimizing XML Schema Languages

As mentioned before, XML is the lingua franca for data exchange on the Internet [1]. Within applications or communities, XML data is usually not arbitrary but adheres to some structure imposed by a schema. The presence of such a schema not only provides users with a global view on the anatomy of the data, but far more importantly, it enables automation and optimization of standard tasks like (*i*) searching, integration, and processing of XML data (cf., e.g., [28, 69, 73, 109]); and, (*ii*) static analysis of transformations (cf., e.g., [5, 56, 74, 87]). Decision problems like equivalence, inclusion and non-emptiness of intersection of schemas, hereafter referred to as *the basic decision problems*, constitute essential building blocks in solutions for the just mentioned optimization and static analysis problems. Additionally, the basic decision problems are fundamental for schema minimization (cf., e.g., [25, 78]). Because of their widespread applicability, it is therefore important to establish the exact complexity of the basic decision problems for the various XML schema languages.

The most common schema languages are DTD, XML Schema [100], and Relax NG [22] and can be modeled by grammar formalisms [85]. In particular, DTDs correspond to context-free grammars with regular expressions at right-hand sides, while Relax NG is usually abstracted by extended DTDs (EDTDs) [88] or equivalently, unranked tree automata [16], defining the regular unranked tree languages. XML Schema is usually abstracted by single-type

shop	$\rightarrow$	regular* & discount-box*
regular	$\rightarrow$	cd
discount-box	$\rightarrow$	cd^{10,12} price
cd	$\rightarrow$	artist & title & price

Figure 8.1: A sample schema using the numerical occurrence and interleave operators. The schema defines a shop that sells CDs and offers a special price for boxes of 10–12 CDs.

EDTDs. As detailed in [75], the relationship between schema formalisms and grammars provides direct upper and lower bounds for the complexity of the basic decision problems.

A closer inspection of the various schema specifications reveals that the above abstractions in terms of grammars with regular expressions is too coarse. Indeed, in addition to the conventional regular expression operators like concatenation, union, and Kleene-star, the XML Schema and the Relax NG specification allow two other operators as well. The XML Schema specification allows to express counting or numerical occurrence constraints which define the minimal and maximal number of times a regular construct can be repeated. Relax NG allows unordered concatenation through the interleaving operator, which is also present in XML Schema in a restricted form. Finally, both DTD and XML Schema require expressions to be deterministic.

We illustrate these additional operators in Figure 8.1. Although the new operators can be expressed by the conventional regular operators, they cannot do so succinctly (see Chapter 4), which has severe implications on the complexity of the basic decision problems.

The goal of this chapter is to study the impact of these counting and interleaving operators on the complexity of the basic decision problems for DTDs, XSDs, and Relax NG. As observed in Section 8.1, the complexity of inclusion and equivalence of $\text{RE}(\#, \&)$ expressions (and subclasses thereof) carries over to DTDs and single-type EDTDs. Therefore, the results in Chapter 5, concerning (subclasses) of $\text{RE}(\#, \&)$ and Section 6.7, concerning deterministic expressions with counting, immediately give us complexity bounds for the basic decision problems for a wide range of subclasses of DTDs and single-type EDTDs. For EDTDs, this immediate correspondence does not hold anymore. Therefore, we study the complexity of the basic decision problems for EDTDs extended with counting and interleaving operators. Finally, we revisit the simplification problem introduced in [77] for schemas with $\text{RE}(\#, \&)$ expressions. This problem is defined as follows: given an extended DTD, can it be rewritten into an equivalent DTD or a single-type EDTD?

Outline. In **Sections 8.1** and **8.2**, we establish the complexity of the basic decision problems for DTDs and single-type EDTDs, and EDTDs, respectively. We discuss simplification in **Section 8.3**.

8.1 Complexity of DTDs and Single-Type EDTDs

In this section, we establish the complexity of the basic decision problems for DTDs and single-type EDTDs with extended regular expressions. This is mainly done by showing that the results on extended regular expressions of Section 5.2 often carry over literally to DTDs and single-type EDTDs.

We call a complexity class $\mathcal{C}$ *closed under positive reductions* if the following holds for every $O \in \mathcal{C}$. Let L' be accepted by a deterministic polynomial-time Turing machine M with oracle O (denoted $L' = L(M^O)$). Let M further have the property that $L(M^A) \subseteq L(M^B)$ whenever $L(A) \subseteq L(B)$. Then L' is also in $\mathcal{C}$. For a more precise definition of this notion we refer the reader to [54]. For our purposes, it is sufficient that important complexity classes like PTIME, NP, CONP, PSPACE, and EXPSPACE have this property, and that every such class contains PTIME. The following two propositions have been shown to hold for standard regular expressions in [75]. However, their proofs carry over literally to all subclasses of RE(#, &).

Proposition 92 ([75])**.** Let $\mathcal{R}$ be a subclass of RE(#, &) and let $\mathcal{C}$ be a complexity class closed under positive reductions. Then the following are equivalent:

(a) INCLUSION for $\mathcal{R}$ expressions is in $\mathcal{C}$.

(b) INCLUSION for DTD($\mathcal{R}$) is in $\mathcal{C}$.

(c) INCLUSION for EDTD$^{\text{st}}$($\mathcal{R}$) is in $\mathcal{C}$.

The corresponding statement holds for EQUIVALENCE.

Clearly, any lower bound on the complexity of EQUIVALENCE and INCLUSION of a class of regular expressions $\mathcal{R}$ also carries over to DTD($\mathcal{R}$) and EDTD$^{\text{st}}$($\mathcal{R}$). Hence the results on the EQUIVALENCE and INCLUSION problem of Section 5.2 and Section 6.7 all carry over to DTDs and single-type EDTDs. For instance, EQUIVALENCE and INCLUSION for EDTD$^{\text{st}}$(#), EDTD$^{\text{st}}$(#,&), and DTD(&) are EXPSPACE-complete, while these problems are in PSPACE for EDTD$^{\text{st}}$(DRE$^{\#}_{S}$).

The previous proposition can be generalized to INTERSECTION of DTDs as well.

Proposition 93 ([75])**.** Let $\mathcal{R}$ be a subclass of RE($\#, \&$) and let $\mathcal{C}$ be a complexity class which is closed under positive reductions. Then the following are equivalent:

(a) INTERSECTION for $\mathcal{R}$ expressions is in $\mathcal{C}$.

(b) INTERSECTION for DTD($\mathcal{R}$) is in $\mathcal{C}$.

Hence, also the results on INTERSECTION of Section 5.2 carry over to DTDs, and any lower bound on INTERSECTION for $\mathcal{R}$ expressions carries over to DTD($\mathcal{R}$) and EDTD$^{\mathrm{st}}$($\mathcal{R}$). Hence, for example INTERSECTION for DTD(#,&) and DTD(DRE$^{\#}_{W}$) are PSPACE-complete. The only remaining problem is INTERSECTION for single-type EDTDs. Although the above proposition does not hold for single-type EDTDs, we can still establish complexity bounds for them. Indeed, INTERSECTION for EDTD$^{\mathrm{st}}$(RE) is EXPTIME-hard and in the next section we will see that even for EDTD($\#$,&) INTERSECTION remains in EXPTIME. It immediately follows that INTERSECTION for EDTD$^{\mathrm{st}}$(#), EDTD$^{\mathrm{st}}$(&), and EDTD$^{\mathrm{st}}$(#,&) is also EXPTIME-complete.

8.2 Complexity of Extended DTDs

We next consider the complexity of the basic decision problems for EDTDs with numerical occurrence constraints and interleaving. Here, we do not consider deterministic expressions as EDTDs are the abstraction of Relax NG, which does not require expressions to be deterministic. As the basic decision problems are EXPTIME-complete for EDTD(RE), the straightforward approach of translating every RE($\#, \&$) expression into an NFA and then applying the standard algorithms gives rise to a double exponential time complexity. By using the NFA($\#, \&$) introduced in Section 5.1, we can do better: EXPSPACE for INCLUSION and EQUIVALENCE, and, more surprisingly, EXPTIME for INTERSECTION.

Theorem 94. (1) EQUIVALENCE and INCLUSION for EDTD(#,&) are in EXPSPACE;

(2) EQUIVALENCE and INCLUSION for EDTD(#) and EDTD(&) are EXPSPACE-hard; and,

(3) INTERSECTION for EDTD(#,&) is EXPTIME-complete.

Proof. (1) We show that INCLUSION is in EXPSPACE. The upper bound for EQUIVALENCE then immediately follows.

First, we introduce some notation. For an EDTD $D = (\Sigma, \Sigma', d, s, \mu)$, we will denote elements of Σ', i.e., types, by τ. We denote by (D, τ) the

EDTD D with start symbol τ. We define the *depth* of a tree t, denoted by $\text{depth}(t)$, as follows: if $t = \varepsilon$, then $\text{depth}(t) = 0$; and if $t = \sigma(t_1 \cdots t_n)$, then $\text{depth}(t) = \max\{\text{depth}(t_i) \mid i \in \{1, \ldots, n\}\} + 1$.

Suppose that we have two EDTDs $D_1 = (\Sigma, \Sigma'_1, d_1, s_1, \mu_1)$ and $D_2 = (\Sigma, \Sigma'_2, d_2, s_2, \mu_2)$. We provide an EXPSPACE algorithm that decides whether $L(D_1) \not\subseteq L(D_2)$. As EXPSPACE is closed under complement, the theorem follows. The algorithm computes a set E of pairs $(C_1, C_2) \in 2^{\Sigma'_1} \times 2^{\Sigma'_2}$ where $(C_1, C_2) \in E$ if and only if there exists a tree t such that $C_j = \{\tau \in \Sigma'_j \mid t \in L((D_j, \tau))\}$ for each $j = 1, 2$. That is, every C_j is the set of types that can be assigned by D_j to the root of t. Or when viewing D_j as a tree automaton, C_j is the set of states that can be assigned to the root in a run on t. Therefore, we say that t *is a witness for* (C_1, C_2). Notice that $t \in L(D_1)$ (respectively, $t \in L(D_2)$) if $s_1 \in C_1$ (respectively, $s_2 \in C_2$). Hence, $L(D_1) \not\subseteq L(D_2)$ if and only if there exists a pair $(C_1, C_2) \in E$ with $s_1 \in C_1$ and $s_2 \notin C_2$.

We compute the set E in a bottom-up manner as follows:

1. Initially, set $E_1 := \{(C_1, C_2) \mid \exists a \in \Sigma, \tau_1 \in \Sigma'_1, \tau_2 \in \Sigma'_2$ such that $\mu_1(\tau_1) = \mu_2(\tau_2) = a$, and for $i = 1, 2$, $C_i = \{\tau \in \Sigma'_i \mid \varepsilon \in d_i(\tau) \wedge \mu_i(\tau) = a\}\}$.

2. For every $k > 1$, E_k is the union of E_{k-1} and the pairs (C_1, C_2) for which there are $a \in \Sigma$, $n \in \mathbb{N}$ and a string $(C_{1,1}, C_{2,1}) \cdots (C_{1,n}, C_{2,n})$ in E^*_{k-1} such that

$$C_j = \{\tau \in \Sigma'_j \mid \mu_j(\tau) = a, \exists b_{j,1} \in C_{j,1}, \ldots, b_{j,n} \in C_{j,n} \\ \text{with } b_{j,1} \cdots b_{j,n} \in d_j(\tau)\}, \text{ for each } j = 1, 2.$$

Let $E := E_\ell$ for $\ell = 2^{|\Sigma'_1|} \cdot 2^{|\Sigma'_2|}$. The algorithm then accepts when there is a pair $(C_1, C_2) \in E$ with $s_1 \in C_1$ and $s_2 \notin C_2$ and rejects otherwise.

We argue that the algorithm is correct. As $E_k \subseteq E_{k+1}$, for every k, it follows that $E_\ell = E_{\ell+1}$. Hence, the algorithm computes the largest set of pairs. The following lemma then shows that the algorithm decides whether $L(D_1) \not\subseteq L(D_2)$. The lemma can be proved by induction on k.

Lemma 95. For every $k \geq 1$, $(C_1, C_2) \in E_k$ if and only if there exists a witness tree for (C_1, C_2) of depth at most k.

It remains to show that the algorithm can be carried out using exponential space. Step (1) reduces to a linear number of tests $\varepsilon \in L(r)$, for some RE(#, &) expressions r which is in PTIME by [64]. Step (3) can be carried out in exponential time, since the size of E is exponential in the input. For step (2), it suffices to argue that, when E_{k-1} is known, it is decidable in EXPSPACE whether a pair (C_1, C_2) is in E_k. As there are only an exponential number of such possible pairs, the result follows. To this end, we need to verify that there exists a string $W = (C_{1,1}, C_{2,1}) \cdots (C_{1,n}, C_{2,n})$ in E^*_{k-1} such that for each $j = 1, 2$,

(A) for every $\tau \in C_j$, there exist $b_{j,1} \in C_{j,1}, \ldots, b_{j,n} \in C_{j,n}$ with $b_{j,1} \cdots b_{j,n} \in d_j(\tau)$; and,

(B) for every $\tau \in \Sigma'_j \setminus C_j$, there do *not* exist $b_{j,1} \in C_{j,1}, \ldots, b_{j,n} \in C_{j,n}$with $b_{j,1} \cdots b_{j,n} \in d_j(\tau)$.

Assume that $\Sigma'_1 \cap \Sigma'_2 = \emptyset$. Let, for each $j = 1, 2$ and $\tau \in \Sigma'_j$, $N(\tau)$ be the NFA($\#, \&$) accepting $d_j(\tau)$. Intuitively, we guess the string W one symbol at a time and compute the set of reachable configurations Γ_τ for each $N(\tau)$.

Initially, Γ_τ is the singleton set containing the initial configuration of $N(\tau)$. Suppose that we have guessed a prefix $(C_{1,1}, C_{2,1}) \cdots (C_{1,m-1}, C_{2,m-1})$ of W and that we guess a new symbol $(C_{1,m}, C_{2,m})$. Then, we compute the set $\Gamma'_\tau = \{\gamma' \mid \exists b \in C_{j,m}, \gamma \in \Gamma_\tau \text{ such that } \gamma \Rightarrow_{N(\tau),b} \gamma'\}$ and set Γ_τ to Γ'_τ. Each set Γ'_τ can be computed in exponential space from Γ_τ. We accept (C_1, C_2) when for every $\tau \in \Sigma'_j$, $\tau \in C_j$ if and only if Γ_τ contains an accepting configuration.

(2) It is shown by Mayer and Stockmeyer [79] and Meyer and Stockmeyer [83] that EQUIVALENCE and INCLUSION are EXPSPACE-hard for RE(&)s and RE(#), respectively. Hence, EQUIVALENCE and INCLUSION are also EXPSPACE-hard for EDTD(&) and EDTD(#).

(3) The lower bound follows from [99]. We argue that the problem is in EXPTIME. Thereto, let, for each $i \in \{1, \ldots, n\}$, $D_i = (\Sigma, \Sigma'_i, d_i, s_i, \mu_i)$ be an EDTD(#,&). We assume w.l.o.g. that the sets Σ'_i are pairwise disjoint. We also assume that the start type s_i never appears at the right-hand side of a rule. Finally, we assume that no derivation tree consists of only the root. For each type $\tau \in \Sigma'_i$, let $N(\tau)$ denote an NFA($\#, \&$) for $d_i(\tau)$. According to Theorem 42, $N(\tau)$ can be computed from $d_i(\tau)$ in polynomial time. We provide an alternating polynomial space algorithm that guesses a tree t and accepts if $t \in L(D_1) \cap \cdots \cap L(D_n)$. As APSPACE = EXPTIME [19], this proves the theorem.

We guess t node by node in a top-down manner. For every guessed node v, the following information is written on the tape of the Turing machine: for every $i \in \{1, \ldots, n\}$, the triple $c_i = (\tau^i_v, \tau^i_p, \gamma^i)$ where τ^i_v is the type assigned to v by grammar D_i, τ^i_p is the type of the parent assigned by D_i, and γ^i is the current configuration $N(\tau^i_p)$ is in after reading the string formed by the left siblings of v. In the following, we say that $\tau \in \Sigma'_i$ is an a-type when $\mu_i(\tau) = a$.

The algorithm proceeds as follows:

1. As for each grammar the types of the roots are given, we start by guessing the first child of the root. That is, we guess an $a \in \Sigma$, and for each $i \in \{1, \ldots, n\}$, we guess an a-type τ^i and write the triple $c_i = (\tau^i, s_i, \gamma_s^i)$ on the tape where γ_s^i is the start configuration of $N(s_i)$.
2. For $i \in \{1, \ldots, n\}$, let $c_i = (\tau^i, \tau_p^i, \gamma^i)$ be the triples on the tape. The algorithm now universally splits into two parallel branches as follows:
   (a) **Downward extension**: When for every i, $\varepsilon \in d_i(\tau^i)$ then the current node can be a leaf node and the branch accepts. Otherwise, guess an $a \in \Sigma$ and for each i, guess an a-type θ^i. Replace every c_i by the triple $(\theta^i, \tau^i, \gamma_s^i)$ and proceed to step (2). Here, γ_s^i is the start configuration of $N(\tau^i)$.
   (b) **Extension to the right**: For every $i \in \{1, \ldots, n\}$, compute a configuration γ'^i for which $\gamma^i \Rightarrow_{N(\tau_p^i),\tau^i} \gamma'^i$. When every γ'^i is a final configuration, then we do not need to extend to the right anymore and the algorithm accepts. Otherwise, guess an $a \in \Sigma$ and for each i, guess an a-type θ^i. Replace every c_i by the triple $(\theta^i, \tau^i, \gamma'^i)$ and proceed to step (2).

We argue that the algorithm is correct. If the algorithm accepts, we have guessed a tree t and, for every $i \in \{1, \ldots, n\}$, a tree t'_i with $\mu_i(t'_i) = t$ and $t'_i \in L(d_i)$. Therefore, $t \in \bigcap_{i=1}^n L(D_i)$. For the other direction, suppose that there exists a tree $t \in \bigcap_{i=1}^n L(D_i)$ and t is minimal in the sense that no subtree t_0 of t is in $\bigcap_{i=1}^n L(D_i)$. Then, there is a run of the above algorithm that guesses t and guesses trees t'_i with $\mu_i(t'_i) = t$. The tree t must be minimal since the algorithm stops extending the tree as soon as possible.

The algorithm obviously uses only polynomial space. □

8.3 Simplification

In this section, we study the simplification problem a bit more broadly than before. Given an EDTD, we are interested in knowing whether it has an equivalent EDTD of a restricted type, i.e., an equivalent DTD or single-type EDTD. In [77], this problem was shown to be EXPTIME-complete for EDTDs with standard regular expressions. We revisit this problem in the context of RE(#, &).

We need a bit of terminology. We say that a tree language L is *closed under ancestor-guarded subtree exchange* if the following holds. Whenever for two trees $t_1, t_2 \in L$ with nodes $u_1 \in \mathrm{Dom}(t_1)$ and $u_2 \in \mathrm{Dom}(t_2)$, $\text{anc-str}^{t_1}(u_1) = \text{anc-str}^{t_2}(u_2)$ implies $t_1[u_1 \leftarrow \text{subtree}^{t_2}(u_2)] \in L$.

We recall the following theorem from [77]:

Theorem 96 (Theorem 7.1 in [77])**.** Let L be a tree language defined by an EDTD. Then the following conditions are equivalent.

(a) L is definable by a single-type EDTD.

(b) L is closed under ancestor-guarded subtree exchange.

We are now ready for the following theorem.

Theorem 97. Given an EDTD(#,&), deciding whether it is equivalent to an EDTD$^{\mathrm{st}}$(#,&) or DTD(#,&) is EXPSPACE-complete.

Proof. We first show that the problem is hard for EXPSPACE. We use a reduction from EQUIVALENCE of RE(#), which is EXPSPACE-complete [83].

Let r_1, r_2 be RE(#) expressions over Σ and let b and s be two symbols not occurring in Σ. By definition $L(r_j) \neq \emptyset$, for $j = 1, 2$. Define $D = (\Sigma \cup \{b, s\}, \Sigma \cup \{s, b^1, b^2\}, d, s, \mu)$ as the EDTD with the following rules:

$$\begin{array}{lcl} s & \rightarrow & b^1 b^2 \\ b^1 & \rightarrow & r_1 \\ b^2 & \rightarrow & r_2, \end{array}$$

where for every $\tau \in \Sigma \cup \{s\}$, $\mu(\tau) = \tau$, and $\mu(b^1) = \mu(b^2) = b$. We claim that D is equivalent to a single-type DTD or a DTD if and only if $L(r_1) = L(r_2)$. Clearly, if r_1 is equivalent to r_2, then D is equivalent to the DTD (and therefore also to a single-type EDTD)

$$\begin{array}{lcl} s & \rightarrow & bb \\ b & \rightarrow & r_1. \end{array}$$

Conversely, suppose that there exists an EDTD$^{\mathrm{st}}$ which defines the language $L(D)$. Towards a contradiction, assume that r_1 is not equivalent to r_2. So, there exists a string w_1 such that $w_1 \in L(r_1)$ and $w_1 \notin L(r_2)$, or $w_1 \notin L(r_1)$ and $w_1 \in L(r_2)$. We only consider the first case, the second is identical. Now, let w_2 be a string in $L(r_2)$ and consider the tree $t = s(b(w_1)b(w_2))$. Clearly, t is in $L(D)$. However, the tree $t' = s(b(w_2)b(w_1))$ obtained from t by switching its left and right subtree is not in $L(D)$. According to Theorem 96, every tree language defined by a single-type EDTD is closed under such an exchange of subtrees. So, this means that $L(D)$ cannot be defined by an EDTD$^{\mathrm{st}}$, which leads to the desired contradiction.

We now proceed with the upper bounds. The following algorithms are along the same lines as the EXPTIME algorithms in [77] for the simplification problem without numerical occurrence or interleaving operators. We first give an EXPSPACE algorithm which decides whether an EDTD is equivalent to an

EDTD$^{\mathrm{st}}$. Let $D = (\Sigma, \Sigma', d, s, \mu)$ be an EDTD. Intuitively, we compute an EDTD$^{\mathrm{st}}$ $D_0 = (\Sigma, \Sigma'_0, d_0, s, \mu_0)$ which is the closure of D under the single-type property. The EDTD$^{\mathrm{st}}$ D_0 has the following properties:

(a) Σ'_0 is in general exponentially larger than Σ';

(b) the RE$(\#,\&)$ expressions in the definition of d_0 are only polynomially larger than the RE$(\#,\&)$ expressions in the definition of d;

(c) $L(D) \subseteq L(D_0)$; and,

(d) $L(D_0) = L(D) \Leftrightarrow D$ is equivalent to a EDTD$^{\mathrm{st}}$.

Hence, D is equivalent to an EDTD$^{\mathrm{st}}$ if and only if $L(D_0) \subseteq L(D)$.

We first show how D_0 can be constructed. We can assume w.l.o.g. that, for each type $a^i \in \Sigma'$, there exists a tree $t' \in L(d)$ such that a^i is a label in t'. Indeed, every *useless* type can be removed from D in a simple preprocessing step. Then, for a string $w \in \Sigma^*$ and $a \in \Sigma$ let types(wa) be the set of all types $a^i \in \Sigma'$, for which there is a tree t and a tree $t' \in L(d)$ with $\mu(t') = t$, and a node v in t such that anc-str$^t(v) = wa$ and the type of v in t' is a^i. We show how to compute types(wa) in exponential time. To this end, we enumerate all sets types(w). Let $s = c^1$. Initially, set $W := \{c\}$, Types$(c) := \{c^1\}$ and $R := \{\{c^1\}\}$. Repeat the following until W becomes empty:

(1) Remove a string wa from W.

(2) For every $b \in \Sigma$, let Types(wab) contain all b^i for which there exists an a^j in Types(wa) and a string in $d(a^j)$ containing b^i. If Types(wab) is not empty and not already in R, then add it to R and add wab to W.

Since we add every set only once to R, the algorithm runs in time exponential in the size of D. Moreover, we have that Types(w) = types(w) for every w, and that $R = \Sigma'_0$.

For each $a \in \Sigma$, let types(D, a) be the set of all nonempty sets types(wa), with $w \in \Sigma^*$. Clearly, each types(D, a) is finite. We next define $D_0 = (\Sigma, \Sigma'_0, d_0, s, \mu_0)$. Its set of types is $\Sigma'_0 := \bigcup_{a \in \Sigma}$ types(D, a). Note that $s \in \Sigma'_0$. For every $\tau \in$ types(D, a), set $\mu_0(\tau) = a$. In d_0, the right-hand side of the rule for each types(wa) is the disjunction of all $d(a^i)$ for $a^i \in$ types(wa), with each b^j in $d(a^i)$ replaced by types(wab).

We show that properties (a)–(d) hold. Since $\Sigma'_0 \subseteq 2^{\Sigma'}$, we immediately have that (a) holds. The RE$(\#,\&)$ expressions that we constructed in D_0 are unions of a linear number of RE$(\#,\&)$ expressions in D, but have types in $2^{\Sigma'}$ rather than in Σ'. Hence, the size of the RE$(\#,\&)$ expressions in D_0 is at most quadratic in the size of D. Finally, we note that it has been shown in Theorem 7.1 in [77] that (c) and (d) also hold.

It remains to argue that it can be decided in EXPSPACE that $L(D_0) \subseteq L(D)$. A direct application of the EXPSPACE algorithm in Theorem 94(1) leads to a 2EXPSPACE algorithm to test whether $L(D_0) \subseteq L(D)$, due to the computation of C_1. Indeed, the algorithm remembers, given the EDTDs $D_0 = (\Sigma, \Sigma'_0, d_0, s_0, \mu_0)$ and $D = (\Sigma, \Sigma', d, s, \mu)$, all possible pairs (C_1, C_2) such that there exists a tree t with $C_1 = \{\tau \in \Sigma'_0 \mid t \in L((D_0, \tau))\}$ and $C_2 = \{\tau \in \Sigma' \mid t \in L((D, \tau))\}$. It then accepts if there exists such a pair (C_1, C_2) with $s_0 \in C_1$ and $s \notin C_2$. However, when we use non-determinism, notice that it is not necessary to compute the entire set C_1. Indeed, as we only test whether there *exist* elements in C_1 in the entire course of the algorithm, we can adapt the algorithm to compute pairs (c_1, C_2), where c_1 is an element of C_1, rather than the entire set. Since NEXPSPACE = EXPSPACE, we can use this adaption to test whether $L(D_0) \subseteq L(D)$ in EXPSPACE.

Finally, we give the algorithm which decides whether an EDTD $D = (\Sigma, \Sigma', d, s, \mu)$ is equivalent to a DTD. We compute a DTD (Σ, d_0, s_d) which is equivalent to D if and only if $L(D)$ is definable by a DTD. Thereto, let for each $a^i \in \Sigma'$, $r_{a,i}$ be the expression obtained from $d(a^i)$ by replacing each symbol b^j in $d(a^i)$ by b. For every $a \in \Sigma$, define $d_0(a) = \bigcup_{a^i \in \Sigma'} r_{a,i}$. Again, it is shown in [77] that $L(D) = L(d_0)$ if and only if $L(D)$ is definable by a DTD. By Theorem 94(1) and since d_0 is of size polynomial in the size of D, this can be tested in EXPSPACE. $\square$

9

Handling Non-Deterministic Regular Expressions

At several places already in this thesis we have discussed the requirement in DTD and XML Schema of regular expressions to be deterministic. Unfortunately, the Unique Particle Attribution (UPA) constraint, as determinism is called in XML Schema, is a highly non-transparent one. The sole motivation for this restriction is backward compatibility with SGML, a predecessor of XML, where it was introduced for the reason of fast unambiguous parsing of content models (without lookahead) [103]. Sadly this notion of unambiguous parsing is a semantic rather than a syntactic one, making it difficult for designers to interpret. Specifically, the XML Schema specification mentions the following definition of UPA:

> A content model must be formed such that during *validation* of an element information item sequence, the particle component contained directly, indirectly or *implicitly* therein with which to attempt to *validate* each item in the sequence in turn can be uniquely determined without examining the content or attributes of that item, and without any information about the items in the remainder of the sequence. [italics mine]

In most books (c.f. [103]), the UPA constraint is usually explained in terms of a simple example rather than by means of a clear syntactical definition. The latter is not surprising as to date there is no known easy syntax for deterministic regular expressions. That is, there are no simple rules a user can apply to define only (and all) deterministic regular expressions. So, when after the schema design process, one or several content models are rejected by the schema checker on account of being nondeterministic, it is very difficult for non-expert[1] users to grasp the source of the error and almost impossible to rewrite the content model into an admissible one. The purpose of the present chapter is to investigate methods for transforming nondeterministic expressions into concise and readable deterministic ones defining either the same language or constituting good approximations. We propose the algorithm SUPAC (Supportive UPA Checker) which can be incorporated in a responsive XSD tester which in addition to rejecting XSDs violating UPA also suggests plausible alternatives. Consequently, the task of designing an XSD is relieved from the burden of the UPA restriction and the user can focus on designing an accurate schema. In addition, our algorithm can serve as a plug-in for any model management tool which supports export to XML Schema format [6].

Deterministic regular expressions were investigated in a seminal paper by Brüggemann-Klein and Wood [17]. They show that deciding whether a given regular expression is deterministic can be done in quadratic time. In addition, they provide an algorithm, that we call BKW^{dec}, to decide whether a regular language can be represented by a deterministic regular expression. BKW^{dec} runs in time quadratic in the size of the minimal deterministic finite automaton and therefore in time exponential in the size of the regular expression. We prove in this chapter that the problem is hard for PSPACE thereby eliminating much of the hope for a theoretically tractable algorithm. We tested BKW^{dec} on a large and diverse set of regular expressions and observed that it runs very fast (under 200ms for expressions with 50 alphabet symbols). It turns out that, for many expressions, the corresponding minimal DFA is quite small and far from the theoretical worst-case exponential size increase. In addition, we observe that BKW^{dec} is fixed-parameter tractable in the maximal number of occurrences of the same alphabet symbol. As this number is very small for the far majority of real-world regular expressions [9], applying BKW^{dec} in practice should never be a problem.

Deciding existence of an equivalent deterministic regular expression or effectively constructing one, are entirely different matters. Indeed, while the decision problem is in EXPTIME, Brüggemann-Klein and Wood [17] provide an algorithm, which we will call BKW, which constructs deterministic regular expressions whose size can be double exponential in the size of the original expression. In this chapter, we measure the size of an expression as the total number of occurrences of alphabet symbols. The first exponential size increase stems from creating the minimal deterministic automaton A_r equivalent to the

[1]In formal language theory.

given nondeterministic regular expression r. The second one stems from translating the automaton into an expression. Although it is unclear whether this double exponential size increase can be avoided, examples are known for which a single exponential blow-up is necessary [17]. We define an optimized version of BKW, called BKW-OPT, which optimizes the second step in the algorithm and can produce exponentially smaller expressions than BKW. Unfortunately, the obtained expressions can still be very large. For instance, as detailed in the experiments section, for input expressions of size 15, BKW and BKW-OPT generate equivalent deterministic expressions of average size 1577 and 394, respectively. To overcome this, we propose the algorithm GROW. The idea underlying this algorithm is that small deterministic regular expressions correspond to small Glushkov automata [17]: indeed, every deterministic regular expression r can be translated in a Glushkov automaton with as many states as there are alphabet symbols in r. Therefore, when the minimal automaton A_r is not Glushkov, GROW tries to extend A_r such that it becomes Glushkov. To translate the Glushkov automaton into an equivalent regular expression, we use the existing algorithm REWRITE [9]. Our experiments show that when GROW succeeds in finding a small equivalent deterministic expression its size is always roughly that of the input expression. In this respect, it is greatly superior to BKW and BKW-OPT. Nevertheless, its success rate is inversely proportional to the size of the input expression (we refer to Section 9.4.2 for details).

Next, we focus on the case when no equivalent deterministic regular expression can be constructed for a given nondeterministic regular expression and an adequate super-approximation is needed. We start with a fairly negative result: we show that there is no smallest super-approximation of a regular expression r within the class of deterministic regular expressions. That is, whenever $L(r) \subsetneq L(s)$, and s is deterministic, then there is a deterministic expression s' with $L(r) \subsetneq L(s') \subsetneq L(s)$. We therefore measure the proximity between r and s relative to the strings up to a fixed length. Using this measure we can compare the quality of different approximations. We consider three algorithms. The first one is an algorithm of Ahonen [3] which essentially repairs BKW by adding edges to the minimal DFA whenever it gets stuck. The second algorithm operates like the first one but utilizes GROW rather than BKW to generate the corresponding deterministic regular expression. The third algorithm, called SHRINK, merges states, thereby generalizing the language, until a regular language is obtained with a corresponding concise deterministic regular expression. For the latter, we again make use of GROW, and of the algorithm KOA-TO-KORE of [8] which transforms automata to concise regular expressions. In our experimental study, we show in which situation which of the algorithms works best.

Finally, based on the experimental assessment, we propose the algorithm SUPAC (supportive UPA checker) for handling non-deterministic regular expressions. SUPAC makes use of several of the aforementioned algorithms and can be incorporated in a responsive XSD checker to automatically deal with the UPA constraint.

Related Work. Although XML is accepted as the de facto standard for data exchange on the Internet and XML Schema is widely used, fairly little attention has been devoted to the study of deterministic regular expressions. We already mentioned the seminal paper by Bruggemann-Klein and Wood [17]. Computational and structural properties were addressed by Martens, Neven, and Schwentick [75]. They show that testing non-emptiness of an arbitrary number of intersections of deterministic regular expressions is PSPACE-complete. Bex et al. investigated algorithms for the inference of regular expressions from a sample of strings in the context of DTDs and XML Schemas [8, 9, 11, 12]. From this investigation resulted two algorithms: REWRITE [9] which transforms automata with n states to equivalent expressions with n alphabet symbols and fails when no such expression exists; and the algorithm KOA-TO-KORE [8, 9] which operates as REWRITE with the difference that it always returns a concise expression at the expense of generalizing the language when no equivalent concise expression exists.

Deciding determinism of expressions containing numerical occurrences was studied by Kilpeläinen and Tuhkanen [66]. The complexity of syntactic subclasses of the deterministic regular expressions with counting has also been considered [43, 44]. In the context of streaming the notion of determinism and k-determinism was used in [68] and [20].

The presented work would clearly benefit from algorithms for regular expression minimization. To the best of our knowledge, no such (efficient) algorithms exist for deterministic regular expressions, for which minimization is in NP. Only a sound and complete rewriting system is available for general regular expressions [95], for which minimization is PSPACE-complete.

Outline. The outline of the chapter is as follows. In **Section 9.1**, we discuss the complexity of deciding determinism of the underlying regular language. In **Section 9.2** and **9.3**, we discuss the construction of equivalent and super-approximations of regular expressions, respectively. We present an experimental validation of our algorithms in **Section 9.4**. We outline a supportive UPA checker in **Section 9.5**.

9.1 Deciding Determinism

The first step in creating a responsive UPA checker is testing whether $L(r)$ is deterministic. Brüggemann-Klein and Wood obtained an EXPTIME algorithm (in the size of the regular expression) which we will refer to as BKW^{dec}:

Theorem 98 ([17])**.** Given a regular expression r, the algorithm BKW^{dec} decides in time quadratic in the size of the minimal DFA corresponding to r whether $L(r)$ is deterministic.

We show that, unless PSPACE = PTIME, there is no hope for a theoretically tractable algorithm.

Theorem 99. Given a regular expression r, the problem of deciding whether $L(r)$ is deterministic is PSPACE-hard.

Proof. We reduce from the CORRIDOR TILING problem, introduced in Section 5.1. However, we restrict ourselves to those tiling instances for which there exists at most one correct corridor tiling. Notice that we can assume this without loss of generality: From the master reduction from Turing Machine acceptance to CORRIDOR TILING in [104], it follows that the number of correct tilings of the constructed tiling system is precisely the number of accepting runs of the Turing Machine on its input word. As the acceptance problem for polynomial space bounded Turing Machines is already PSPACE-complete for *deterministic* machines, we can assume w.l.o.g. that the input instance of CORRIDOR TILING has at most one correct corridor tiling.

Now, let T be a tiling instance for which there exists at most one correct tiling. We construct a regular expression r, such that $L(r)$ is deterministic if and only if there does not exist a corridor tiling for T. Before giving the actual definition of r, we give the language it will define and show this is indeed deterministic if and only if CORRIDOR TILING for T is false. We encode corridor tilings by a string in which the different rows are separated by the symbol \$, that is, by strings of the form

$$\$R_1\$R_2\$\cdots\$R_m\$$$

in which each R_i represents a row and is therefore in X^n. Moreover, R_1 is the bottom row and R_n is the top row.

Then, let $\Sigma = X \uplus \{a, \$, \#\}$ and for a symbol $b \in \Sigma$, let Σ_b denote $\Sigma \setminus \{b\}$. Then, $L(r) = \Sigma^* \setminus \{w_1\#w_2 \mid w_1$ encodes a valid tiling for T and $w_2 \in \Sigma_{\#}^*\Sigma_{a,\#}\Sigma_{\#}\}$. First, if there does not exist a valid tiling for T, then $L(r) = \Sigma^*$ and thus $L(r)$ is deterministic. Conversely, if there does exist a valid corridor tiling for T, then by our assumption, there exists exactly one. A DFA for $L(r)$

is graphically illustrated in Figure 9.1. Notice that this DFA is the minimal DFA if and only if w_1 exists. By applying the algorithm of Brüggemann-Klein and Wood (Algorithm 4), it is easily seen that $L(r)$ is not deterministic. Indeed, Algorithm 4 gets immediately stuck in line 17, where it sees that the gates in the orbit consisting of the three rightmost states in Figure 9.1 are not all final states. Hence, this minimal DFA does not satisfy the orbit property.

Our regular expression r now consists of the disjunction of the following regular expressions:[2]

- $\Sigma^*\#\Sigma^*\#\Sigma^* + \Sigma^*_{\#}$: This expression detects strings that do not have exactly one occurrence of #.
- $\Sigma^*\#\Sigma?$: This expression detects strings that have # as last or second to last symbol.
- $\Sigma^*_{\#}\Sigma^* a\Sigma$: This expression detects strings that have a as second to last symbol.
- $\Sigma^* a\Sigma^*\#\Sigma^*$: This expression detects strings that have an a before the #-sign.
- $\Sigma_{\$}\Sigma^* + \Sigma^*\Sigma_{\$}\#\Sigma^*$: This expression detects strings that do not have a \$-sign as the first or last element of their encoding.
- $\Sigma^*\$\Sigma_{\$}^{[0,n-1]}\$\Sigma^*\#\Sigma^* + \Sigma^*\$\Sigma_{\$}^{[n+1,n+1]}\Sigma^*_{\$}\$\Sigma^*\#\Sigma^*$: This expression detects all string in which a row in the tiling encoding is too short or too long.
- $\Sigma^* x_1 x_2 \Sigma^*\#\Sigma^*$, for every $x_1, x_2 \in X, (x_1, x_2) \notin H$: These expressions detect all violations of horizontal constraints in the tiling encoding.
- $\Sigma^* x_1 \Sigma^n x_2 \Sigma^*\#\Sigma^*$, for every $x_1, x_2 \in X, (x_1, x_2) \notin V$: These expressions detect all violations of vertical constraints in the tiling encoding.
- $\Sigma^{i+1}\Sigma_{\bar{b}_i}\Sigma^*\#\Sigma^*$ for every $1 \leq i < n$: These expressions detect all tilings which do not have $\bar{b}$ as the bottom row in the tiling encoding.
- $\Sigma^*\Sigma_{\bar{t}_i}\Sigma^{n-i}\#\Sigma^*$ for every $1 \leq i < n$: These expressions detect all tilings which do not have $\bar{t}$ as the top row in the tiling encoding.

Finally, it is easily verified that $L(r)$ is defined correctly. □

It is unclear whether the problem itself is in PSPACE. Simply guessing a deterministic regular expression s and testing equivalence with r does not work as the size of s can be exponential in the size of r (see also Theorem 102).

Next, we address the problem from the viewpoint of parameterized complexity [34], where an additional parameter k is extracted from the input r. Then, we say that a problem is *fixed parameter tractable* if there exists a computable function f and a polynomial g such that the problem can be solved in

[2]Notice that r itself does not have to be deterministic.

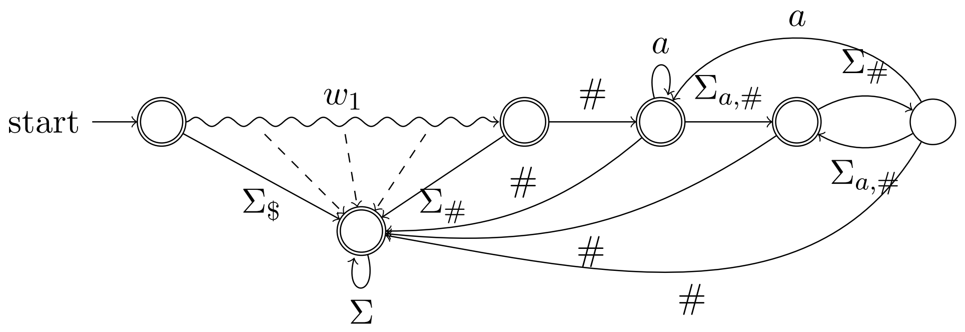


Figure 9.1: DFA for $L(r)$ in the proof of Theorem 99.

time $f(k) \cdot g(|r|)$. Intuitively, this implies that, if k is small and f is reasonable, the problem is efficiently solvable. We now say that an expression r is a *k-occurrence regular expression (k-ORE)* if every alphabet symbol occurs at most k times in r. For example, $ab(a^* + c)$ is a 2-ORE because a occurs twice.

Proposition 100. Let r be a k-ORE. The problem of deciding whether $L(r)$ is deterministic is fixed parameter tractable with parameter k. Specifically, its complexity is $\mathcal{O}(2^{k^2}|r|^2)$.

Proof. Let r be a k-ORE. By applying a Glushkov construction (see, e.g., [17]), followed by a subset construction, it is easy to see that the resulting DFA has at most $2^k \cdot |\Sigma|$ states. By Theorem 98, the result follows. □

This result is not only of theoretical interest. It has already often been observed that the vast majority of regular expressions occurring in practice are k-OREs, for $k = 1, 2, 3$ (see, e.g., [77]). Hence, this result implies that in practice the problem can be solved in polynomial time.

Corollary 101. For any fixed k, the problem of deciding whether the language defined by a k-ORE is deterministic is in PTIME.

9.2 Constructing Deterministic Expressions

Next, we focus on constructing equivalent deterministic regular expressions. Unfortunately, the following result by Brüggemann-Klein and Wood already rules out a truly efficient conversion algorithm:

Theorem 102 ([17]). For any $n \in \mathbb{N}$, there exists a regular expression r_n of size $\mathcal{O}(n)$ such that any deterministic regular expression defining $L(r_n)$ is of size at least 2^n.

Algorithm 2 Algorithm GROW, with pool size P and expansion size E.

```
Require: $P$, $E \in \mathbb{N}$, minimal DFA $A = (Q, \Sigma, \delta, q_0, F)$ with $L(A)$ determin-
    istic,
Ensure: Det. reg. exp. $s$ with $L(s) = L(A)$, if successful
    for $i = 0$ to $E$ do
 2:   Generate at most $P$ non-isomorphic DFAs $B$ s.t.
                              $L(B) = L(A)$ and $B$ has $|Q| + i$ states
 4:   for each such $B$ do
        if REWRITE $(B)$ succeeds then
 6:       return REWRITE $(B)$
    if $i > E$ then fail
```

9.2.1 Growing automata

We first present GROW as Algorithm 2, which is designed to produce concise deterministic expressions. The idea underlying this algorithm is that the Glushkov construction [17] transforms small deterministic regular expressions to small deterministic automata with as many states as there are alphabet symbols in the expression. The minimization algorithm eliminates some of these states, complicating the inverse Glushkov-rewriting from DFA to deterministic regular expression. By expanding the minimal automaton, GROW tries to recuperate the eliminated states. The algorithm REWRITE of [9] succeeds when the modified automaton can be obtained from a deterministic regular expression by the Glushkov construction and assembles this expression upon success. As their are many DFAs equivalent to the given automaton A, we only enumerate non-isomorphic expansions of A up to a given number of extra states E. However, the number of generated non-isomorphic DFAs can explode quickly. Therefore, the algorithm is also given a given pool size P which restricts the number of DFAs of each size which are generated.

Nonwithstanding the harsh brute force flavor of GROW, we show in our experimental study that the algorithm can be quite effective.

9.2.2 Enumerating Automata

The GROW algorithm described above uses an enumeration algorithm as a subroutine. In this section, we describe this algorithm which, given a minimal DFA M and a size k, efficiently enumerates all non-isomorphic DFAs equivalent to M of size at most k. Thereto, we first provide some definitions.

Throughout this section, we assume that any DFA is trimmed, i.e., contains no useless transitions. For a DFA A, we always denote its set of states by Q_A, transitions by δ_A, initial state by q_A, and set of final states by F_A. Finally, we assume that the set of states Q of any DFA consists of an initial segment of the integers. That is, if $|Q| = n$, then $Q = \{0, \dots, n-1\}$.

Let A and B be DFAs. Then, A and B are *isomorphic*, denoted $A \cong B$, if there exists a bijection $f : Q_A \to Q_B$, such that (1) $f(q_A) = q_B$, (2) for all $q \in Q_A$, we have $q \in F_A$ if and only if $f(q) \in F_B$, and (3) for all $q, p \in Q_A$ and $a \in \Sigma$, it holds that $(q, a, p) \in Q_A$ if and only if $(f(q), a, f(p)) \in Q_B$. We then also say that f *witnesses* $A \cong B$. Notice that, because we are working with DFAs, there is at most one such bijection which, moreover, can easily be constructed. Indeed, we must set $f(q_A) = q_B$. Then, because q_A has at most one outgoing transition for each alphabet symbol this, in turn, defines $f(q)$ for all q to which q_A has an outgoing transition. By continuing recursively in this manner, one obtains the unique bijection witnessing $A \cong B$

Further, we define a function $\mathrm{bf} : Q_A \to [1, |Q_A|]$ which assigns to every state $q \in Q_A$ its number in a breadth-first traversal of A, starting at q_A. Here, we fix an order on the alphabet symbols (say, lexicographic ordering), and require that in the breadth-first traversal of the DFA transitions are followed in the order of the symbol by which they are labeled. This ensures that bf is unambiguously defined. We then say that A is *canonical* if $\mathrm{bf}(i) = i$, for all $i \in Q_A$ (recall that the state labels are always an initial part of the natural numbers). That is, A is canonical if and only if the names of its states correspond to their number in the breadth-first traversal of A. The latter notion of canonical DFAs essentially comes from [4], although there it is presented in terms of a string representation. The following simple lemma is implicit in their paper.

Lemma 103. 1. Let A be a DFA. Then, there exists a canonical DFA B such that $A \cong B$.

2. Let A, B be canonical DFAs. Then, $A = B$ if and only if $A \cong B$.

Now, let M be a minimal DFA, and k an integer and define $S = \{A \mid L(A) = L(M), |Q_A| \leq k, A \text{ is canonical}\}$. Then, by the above lemma, S contains exactly one isomorphic copy of any DFA A equivalent to M of size at most k. Hence, given M and k, the goal of this section reduces to give an algorithm which computes the set S.

To do so efficiently, we first need some insight in the connection between a minimal DFA and its larger equivalent versions. If A is a DFA, and $q \in Q_A$, recall that A^q denotes the DFA A in which q is the initial state. The following lemma then states a few basic facts about minimal DFAs (see, e.g., [55]).

Lemma 104. Let A be a DFA, and M its equivalent minimal DFA.

- For all $q, p \in Q_M$, if $q \neq p$, then $L(M^q) \neq L(M^p)$.

- For all $q \in Q_A$, there exists a (unique) $p \in Q_M$, such that $L(A^q) = L(M^p)$.

Let minimal : $Q_A \to Q_M$ be the function which takes the state $q \in Q_A$ to $p \in Q_M$, when $L(A^q) = L(M^p)$. According to the above lemma, this function is well-defined. We also say that q is a *copy* of p. Hence, for each state $p \in Q_M$, there is at least one copy in A, although there can be more. Further, for any $p' \in Q_M$, and $a \in \Sigma$, if $(p, a, p') \in \delta_M$, then there exists a $q' \in Q_A$, such that minimal$(q') = p'$, and $(q, a, q') \in Q_A$. That is, any $q \in Q_A$ is a copy of minimal$(q) \in Q_M$, and, furthermore, q's outgoing transitions are copies of those of minimal(q), again to copies of the appropriate target states in M. Using this knowledge, Algorithm 3 enumerates the desired automata. For ease of exposition, it is presented as a non-deterministic algorithm in which each trace only outputs one DFA. It can easily be modified, using recursion, to an algorithm outputting all these automata, and hence the desired set.

Theorem 105. Given a minimal DFA M, and $k \geq |Q_M|$, the set S of all automata returned by ENUMERATE, consists of pairwise non-isomorphic DFAs of size at most k. Further, for any DFA B of size at most k and equivalent to M, there is a $B' \in S$ such that $B \cong B'$.

Proof. We first argue that given M, and $k \geq |Q_M|$, ENUMERATE returns the set $S = \{A \mid |Q_A| < k, L(A) = L(M), A \text{ is canonical}\}$. Due to Lemma 103, the set S satisfies the criteria of the theorem.

Let us first show that ENUMERATE only produces automata in this set S, i.e. any DFA A generated by ENUMERATE has $|Q_A| < k$, is equivalent to M, and is canonical. The reason that A is canonical is because the enumeration explicitly follows the breadth-first traversal of the automaton it is generating. It is equivalent to M because for any state $q \in Q_A$, we maintain the corresponding state minimal$(q) \in Q_M$ and give q outgoing transitions to copies of the states where minimal(q) also has transitions to. This guarantees that for all $q \in Q_A$, we have $L(A^q) = L(M^{\text{minimal}(q)})$, and, in particular $L(A) = L(A^{q_a}) = L(M^{q_M}) = L(M)$. Finally, $|Q_A| < k$, due to the test on Line 12, which quarantees that never more than k states will be generated.

Conversely, we must show that any automaton in S is generated by ENUMERATE, i.e. any canonical A equivalent to M with at most k states is generated by ENUMERATE. Let A be such a DFA. By Lemma 104 we know it consists of a number of copies of each state of M which are linked to each other in a manner consistent with M. Given this information, ENUMERATE follows all possible breadth-first paths generating such DFAs, and hence also generates A. □

Algorithm 3 Algorithm ENUMERATE.

```
Require: Minimal DFA $M = (Q_M, \delta_M, q_M, F_M)$, and $k \geq |Q_M|$
Ensure: Canonical DFA $A$, with $L(A) = L(M)$ and $|Q_A| \leq k$.
    $Q_A, \delta_A, F_A \leftarrow \emptyset$
 2: Queue $P \leftarrow \emptyset$
    $q_A \leftarrow 0$
 4: Push $q_A$ onto $P$
    $Q_A \leftarrow Q_A \uplus \{q_A\}$
 6: Set minimal$(q_A) = q_M$
    while $P \neq \emptyset$ do
 8:   $q \leftarrow P$.Pop
      $p \leftarrow$ minimal$(q)$
10:   for $a \in \Sigma$ do
        if $\exists p' \in Q_M : (p, a, p') \in \delta_M$ then
12:       Choose $q'$ non-deterministically such that either $q' = |Q_A|$ (if $|Q_A| <$
    $k$), or such that $q' < |Q_A|$ and minimal$(q') = p'$.
          If no such $q'$ exists: fail.
14:       if $q' = |Q_A|$ then
            Push $q'$ onto $P$
16:         Set minimal$(q') = p'$
            $Q_A \leftarrow Q_A \uplus q'$
18:       $\delta_A \leftarrow \delta_A \uplus (q, a, q')$
          if $p' \in F_M$ then
20:         $F_A \leftarrow F_A \cup q'$
    return $A$
```

We conclude by noting that, by using a careful implementation, the algorithm actually runs in time linear in the total size of the output. Thereto, the algorithm should never fail, but this can be accomplished by maintaining some additional information (concerning the number of states which must still be generated) which allows to avoid non-deterministic choices which will lead to failure.

9.2.3 Optimizing the BKW-Algorithm

Next, we discuss Brüggemann-Klein and Woods BKW algorithm and then present a few optimizations to generate smaller expressions.

First, we need some terminology. Given a DFA A, a symbol a is A-*consistent* if there is a unique state $w(a)$ in A such that all final states of A have an a-transition to $w(a)$. We call $w(a)$ the *witness state* for a. A set S is A-consistent if each element in S is A-consistent. The S-*cut* of A, denoted

Algorithm 4 The BKW-Algorithm.

```
Require: Minimal DFA $A = (Q, \Sigma, \delta, q_0, F)$
Ensure: Det. reg. exp. $s$ with $L(s) = L(A)$
   if $A$ has only one state $q$ and no transitions then
     if $q$ is final then return $\varepsilon$
     else return $\emptyset$
   else if $A$ has precisely one orbit then
     $S \leftarrow A$-consistent symbols
     if $S = \emptyset$ then fail
     else return $\text{BKW}(A_S) \cdot \big(\bigcup_{a \in S} a \cdot \text{BKW}(A_S^{w(a)})\big)^*$
   else
     if $A$ has the orbit property then
       for all $a$ s.t. Orbit($q_0$) has outgoing $a$-transition do
         $q_a \leftarrow$ unique target state of these $a$-transitions
       $A_{q_0} \leftarrow$ orbit automaton of $q_0$
       if $A_{q_0}$ contains a final state then
         return $\text{BKW}(A_{q_0}) \cdot \big(\bigcup_{a \in \Sigma}(a \cdot \text{BKW}(A^{q_a}))\big)?$
       else
         return $\text{BKW}(A_{q_0}) \cdot \bigcup_{a \in \Sigma}(a \cdot \text{BKW}(A^{q_a}))$
     else fail
```

by A_S, is the automaton obtained from A by removing, for each $a \in S$, all a-transitions that leave a final state of A. Given a state q of A, A^q is the automaton obtained from A by setting its initial state to q and restricting its state set to the states reachable from q. For a state q, the *orbit of* q, denoted Orbit(q), is the strongly connected component of A that contains q. We call q a *gate* of Orbit(q) if q is final, or q is the source of a transition that has a target outside Orbit(q).

We say that A has *the orbit property* if, for every pair of gates q_1, q_2 in the same orbit the following properties hold:

1. q_1 is final if and only if q_2 is final; and,
2. for all $a \in \Sigma$ and states q outside the orbit of q_1 and q_2, there is a transition (q_1, a, q) if and only if there is a transition (q_2, a, q).

Given a state q of A, the *orbit automaton* of q, denoted by A_q, is obtained from A by restricting its state set to Orbit(q), setting its initial state to q and by making the gates of Orbit(q) its final states.

The BKW-Algorithm is then given as Algorithm 4. For a regular expression r, the algorithm is called with the minimal complete DFA A accepting $L(r)$

and then recursively constructs an equivalent deterministic expression when one exists and fails otherwise. Algorithm 4 can fail in two places: (1) in line 6, when the set of A-consistent symbols is empty and (2) in line 17, if A does not have the orbit property. Notice that, if A has the orbit property, the unique state q_a on line 11 can always be found. The correctness proof is non-trivial and can be found in [17]. BKW runs in time double exponential in the size of the nondeterministic regular expression. The first exponential arises from converting the given regular expression to a DFA, the second one from branching in the lines 7, 14, and 16. The generated expressions can therefore be quite large. As Algorithm 4 was not designed with conciseness of regular expressions in mind, we therefore propose three optimizations resulting in smaller expressions.

To this end, by slight abuse of notation, let first(A) denote the set $\{a \mid \exists w \in \Sigma^*, aw \in L(A)\}$, i.e., the set of possible first symbols in a string in $L(A)$. We adapt the lines 7, 14, and 16 in Algorithm 4 in the way described below and refer to the modified algorithm as BKW-OPT.

$\boxed{\textit{line 7}}$ Now, A consists of one orbit and S is the set of A-consistent symbols:

- If $L(A_S) = L(A_S^{w(a)})$ for all $a \in S$, $\varepsilon \in L(A_S)$, and $\text{first}(A_S) \cap S = \emptyset$, then return $((S + \varepsilon) \cdot \text{BKW}(A_S))^*$.

- Else, partition S into equivalence classes $S_1, \ldots, S_n$ where for $a, b \in S$, a is equivalent to b if and only if $w(a) = w(b)$. Furthermore, let, for each $i \in \{1, \ldots, n\}$, a_i be an arbitrary but fixed element from S_i. Then, return $\text{BKW}(A_S) \cdot \big(\bigcup_{1 \leq i \leq n} S_i \cdot \text{BKW}(A_{S_i}^{w(a_i)})\big)^*$.

$\boxed{\textit{line 14 and 16}}$ If A consists of more than one orbit, we can view A as an acyclic DFA when considering every orbit as an atomic subautomaton. We therefore define the acyclic DFA summary automaton summ(A) of A where every state corresponds to a unique orbit. As these automata are usually quite small, we subsequently apply GROW to obtain a concise regular expression over an alphabet consisting of Σ-symbols and orbit identifiers. We then replace each orbit identifier by its corresponding, recursively obtained, deterministic expression.

Before defining summary automata formally, we present an example. Figure 9.2(a) illustrates a DFA A with three orbits: $\{1\}$, $\{2, 3, 4\}$, and $\{5\}$. Orbit $\{2, 3, 4\}$ has two possible entry points: states 2 (with an a-transition) and 3 (with the b- and c-transitions). For each such entry point we have a state in the summary automaton. Figure 9.2(b) presents the summary automaton summ(A).

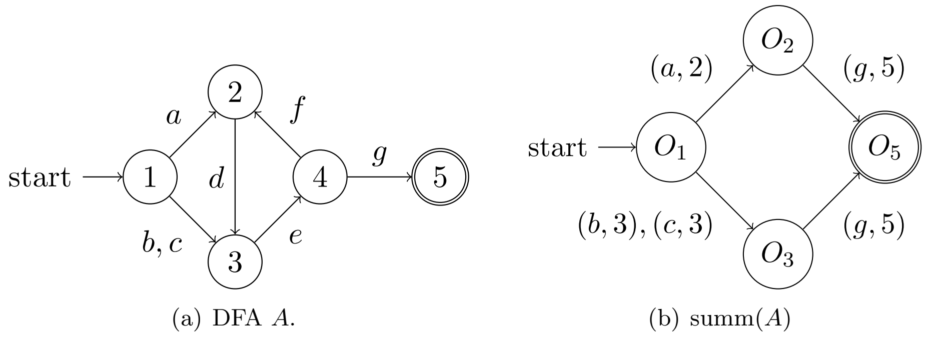


Figure 9.2: A DFA and its summary automaton.

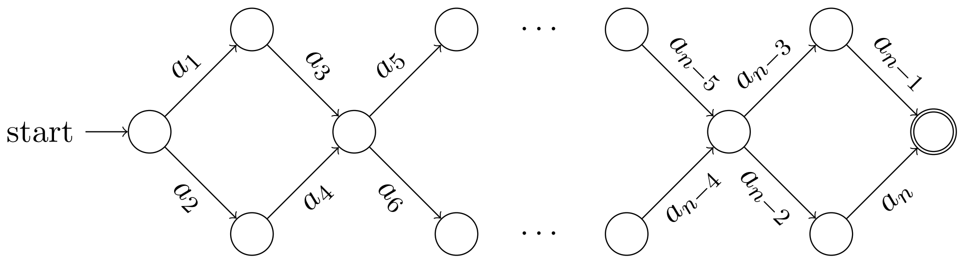


Figure 9.3: Class of DFAs for which our optimization improves exponentially over the BKW algorithm.

Formally, let $A = (Q, \delta, q_0, F)$ over alphabet Σ. Then we define summ(A) as a DFA $(Q^s, \delta^s, q_0^s, F^s)$ over alphabet $\Sigma^s \subseteq \Sigma \times Q$. In particular, for each transition $(q_1, a, q_2) \in \delta$ where $\text{Orbit}(q_1) \neq \text{Orbit}(q_2)$, we have $(a, q_2) \in \Sigma^s$. The state set Q^s is defined as $\{O_q \mid \text{there is a transition } (p, a, q) \in \delta \text{ for } p \text{ outside } \text{Orbit}(q)\}$. Furthermore, we define $q_0^s := O_{q_0}$, $F^s := \{O_p \in Q^s \mid \text{Orbit}(p) \cap F \neq \emptyset\}$, and $(O_{q_1}, (a, q_2), O_{q_2}) \in \delta^s$ if and only if $\text{Orbit}(q_1) \neq \text{Orbit}(q_2)$ and there exists a $q_1' \in \text{Orbit}(q_1)$ such that $(q_1', a, q_2) \in \delta$. Notice that, if A is a DFA fulfilling the orbit property, all outgoing transitions of each orbit go to the same witness state. Therefore, summ(A) is also a DFA.

To find a small regular expression for the multiple orbits case, we make use of the deterministic expressions r_q which are obtained by applying BKW-OPT recursively to the orbit automata A_q. We run GROW on summ(A) to find a small deterministic expression for $L(\text{summ}(A))$. If we find one, we obtain the deterministic expression for $L(A)$ by replacing each symbol (a, q) by $a \cdot r_q$.

Notice that this optimization potentially generates exponentially smaller regular expressions than the BKW algorithm. Consider the family of DFAs of Figure 9.3. The summary DFAs for these automata are equal to the DFAs themselves. While the BKW algorithm would essentially unfold this DFA and return a regular expression of size at least 2^n, GROW would return the expression $(a_1a_3 + a_2a_4) \cdots (a_{n-3}a_{n-1} + a_{n-2}a_n)$, which is linear in n.

It is shown in [53] that there are acyclic DFAs whose smallest equivalent regular expression is of superpolynomial size: $\Omega(n^{\log n})$ for n the size of the DFA. As acyclic DFAs define finite languages and finite languages are deterministic, the result transfers to deterministic regular expressions. Hence, it is impossible for GROW to always return a (small) result. Therefore, when GROW does not find a solution we just apply one non-optimized step of the BKW algorithm (i.e., return line 14/16 of Algorithm 4). However, in our experiments we noticed that this almost never happened (less than 1% of the total calls to GROW did not return an expression).

9.3 Approximating Deterministic Regular Expressions

When the regular language under consideration is not deterministic, we can make it deterministic at the expense of generalizing the language. First, we show that there is no best approximation.

9.3.1 Optimal Approximations

An expression s is a *deterministic super-approximation* of an expression r when $L(r) \subseteq L(s)$ and s is deterministic. In the sequel we will just say approximation rather than super-approximation. Then, we say that s is an *optimal deterministic approximation* of r, if $L(r) \subseteq L(s)$, and there does not exist a deterministic regular expression s' such that $L(r) \subseteq L(s') \subsetneq L(s)$. That is, an approximation is optimal if there does not exist another one which is strictly better. Unfortunately, we can show that no such optimal approximation exists:

Theorem 106. Let r be a regular expression, such that $L(r)$ is not deterministic. Then, there does not exist an optimal deterministic approximation of r.

Proof. We show in Lemma 107 below that for every deterministic language L and string $w \in L$, the language $L \setminus \{w\}$ is also deterministic. Now, suppose, towards a contradiction, that an optimal deterministic approximation s exists. Then, since $L(r)$ is not deterministic, $L(r) \subsetneq L(s)$ and thus there exists some string w with $w \in L(s)$ but $w \notin L(r)$. But then, for the language $L_w = L(s) \setminus \{w\}$, we have that $L(r) \subseteq L_w$ and, by Lemma 107, L_w is also deterministic. This gives us the desired contradiction. □

Lemma 107. For every deterministic language L and string $w \in L$, the language $L \setminus \{w\}$ is also deterministic.

Proof. By $|u|$ we denote the length of string u. We define the prefix-language $L^{\leq |w|} = \{u \in L \mid |u| < |w|\} \cup \{u \in \Sigma^* \mid |u| = |w|, \exists v.uv \in L\}$. As $L^{\leq k}$ is a finite language, one can easily construct a deterministic regular expression for it consisting of nested disjunctions. For instance, the set $L^{\leq |w|} = \{aab, ab, baa, bba\}$ can be defined by $aa(b+\varepsilon)+b(aa+ba)$. Denote the resulting expression by r. Further, we note that deterministic regular languages are closed under *derivatives* [17]. Specifically, for a string u, the u-derivative of a regular expression s is a regular expression s' that defines the set $\{v \mid uv \in L(s)\}$. For every string $u \in L^{\leq |w|}$ with $|u| = |w|$ and $u \neq w$, let r_u be a deterministic expression defining the u-derivative of L. Notice that the u-derivative can be ε. Now for $u = w$, let r_w define the w-derivative of L minus ε. It is shown in [17] (Theorem D.4) that, for every deterministic regular language L', $L' - \{\varepsilon\}$ is also deterministic. Hence, r_w can also be written as a deterministic regular expression. Now, the expression defining $L \setminus \{w\}$ is obtained by adding, for each $u \in L^{\leq |w|}$ with $|u| = |w|$, r_u after the last symbol of u in r. □

As finite languages are always deterministic, Theorem 106 implies that every approximation defines infinitely more strings than the original expression. Furthermore, one can prove analogously that an optimal *under-approximation* of a non-deterministic regular expression r also does not exist. That is, a deterministic regular expression s such that $L(s) \subseteq L(r)$ for which there is no deterministic s' with $L(s) \subsetneq L(s') \subseteq L(r)$.

9.3.2 Quality of the approximation

Motivated by the above discussion, we will compare sizes of regular languages by only comparing strings up to a predefined length.

Thereto, for an expression r and a natural number ℓ, let $L^{\ell}(r)$ be the subset of strings in $L(r)$ with length exactly ℓ. For regular expressions r and s with $L(r) \subseteq L(s)$, define the *proximity* between r and s, as

$$\text{proximity}(r, s) := \frac{1}{k} \sum_{\ell=1}^{k} \frac{|L^{\ell}(r)|+1}{|L^{\ell}(s)|+1}$$

for $k = \max\{2|r|+1, 2|s|+1\}$. The proximity is always a value between 0 and 1. When the proximity is close to 1, the size of the sets $L^{\ell}(s) \setminus L^{\ell}(r)$ is small, and the quality of approximation is excellent.

Although the above measure provides us with a tool to compare proximity of regular languages, we cannot simply search for a deterministic expression which performs best under this measure. It is also important that the obtained expression is small and thus readable. Indeed, a user might well favor an expression which is understandable, but constitutes only a rough approximation, over one which is very close to the original expression, but is completely incomprehensible due to its large size.

In conclusion, a valid approximation s for an expression r, is a deterministic expression constituting a good tradeoff between (1) a large value for proximity(r, s) and (2) a small size $|s|$. The heuristics in the following section will try to construct approximations which fit these requirements.

9.3.3 Ahonen's Algorithm

In the previous section, we have seen that the BKW-algorithm will translate a DFA into a deterministic expression, and will fail if no such equivalent deterministic expression exists. Ahonen's algorithm [3] is a first method that constructs a deterministic regular expression at the expense of generalizing the target language. It essentially runs the BKW$^{\text{dec}}$-algorithm, the decision variant of BKW which does not produce an output expression (cf. Theorem 98), until it fails, and subsequently *repairs* the DFA by adding transitions, making states final, or merging states, in such a manner that BKW$^{\text{dec}}$ can continue. In the end, a DFA is produced defining a deterministic language. The corresponding deterministic regular expression is then obtained by running BKW. Ahonen's algorithm for obtaining a DFA defining a deterministic language is presented in Algorithm 5.[3] By AHONEN-BKW we then denote the application of BKW on the result of AHONEN.

AHONEN proceeds by merging states. We explain in more detail how we can merge two states in a DFA $A = (Q, \delta, q_0, F)$. For an example, see Figure 9.4(c) and 9.4(d), where states 2 and 4 are merged into a new state $\{2, 4\}$. For ease of exposition, we assume that states in Q are sets. Initially, all sets in Q are singletons (e.g., $\{2\}$, $\{4\}$) and by merging such states we obtain non-singletons (e.g., $\{2, 4\}$). Let q_1 and q_2 be the two states to be merged into a new state $q_M := q_1 \cup q_2$. We denote by Q_{new} the state set of A after this merge operation (analogously for δ_{new},$q0_{\text{new}}$, and F_{new}). We assume that $q_1, q_2 \in Q$ and $q_M \notin Q$. Then, $Q_{\text{new}} := (Q \cup \{q_M\}) \setminus \{q_1, q_2\}$. Furthermore, $q_M \in F_{\text{new}}$ if and only if $q_1 \in F$ or $q_2 \in F$. Analogously, $q0_{\text{new}}$ is the unique set $p \in Q_{\text{new}}$ such that $q_0 \in p$. The transitions are adapted by removing all transitions containing q_1 or q_2, but redirecting all these transitions to q_M. For instance, when $(q, a, q_1) \in \delta$, we have $(q, a, q_M) \in \delta_{\text{new}}$. As long as the obtained automaton is not deterministic, we choose non-deterministic transitions (p, a, q_1) and (p, a, q_2) and continue

[3]The algorithm we present slightly differs from Ahonen's original algorithm as the original algorithm is slightly incorrect. We briefly discuss this at the end of this section.

Algorithm 5 An adaptation of Ahonen's repair algorithm: the AHONEN algorithm.

```
Require: DFA $A = (Q, \Sigma, \delta, q_0, F)$
Ensure: DFA $B$ such that $L(A) \subseteq L(B)$
    $S \leftarrow A$-consistent symbols
 2: if $A$ has only one state $q$ then
      if $q$ is final then return $\varepsilon$
 4:   else return $\emptyset$
    else if $A$ has precisely one orbit then
 6:   if $S = \emptyset$ then
        Choose an $a$ s.t. $(q, a, q_1) \in \delta$ for $q$ final
 8:     for all $p \in F$ do
          add $(p, a, q_1)$ to $\delta$
10:       if $(p, a, q_2) \in \delta$ for $q_2 \neq q_1$ then
            Merge($q_1$,$q_2$)
12:     $S \leftarrow \{a\}$
    ForceOrbitProperty ($A_S$)
14: for each orbit $O$ of $A_S$ do
      Choose an arbitrary $q \in O$
16:   AHONEN ($(A_S)_q$)
```

merging states until it is deterministic again. We denote this recursive merging procedure in Algorithms 5 and 6 by MERGE.

AHONEN repairs the automaton A in two possible instances where BKW$^{\text{dec}}$ gets stuck. If A has one orbit but no A-consistent symbols, AHONEN simply chooses a symbol a and adds transitions to force A-consistency. If A has more than one orbit, but does not fulfill the orbit property, then AHONEN calls FORCEORBITPROPERTY (Algorithm 6) to add final states and transitions until A fulfills it.

Example 108. We illustrate the algorithm on the regular expression $(aba + a)^+b$. The minimal DFA A is depicted in Figure 9.4(a). As there are no A-consistent symbols, we have that $S = \emptyset$. As there are three orbits ($\{1\}$, $\{2, 3, 4\}$, and $\{5\}$), the algorithm immediately calls ForceOrbitProperty on A, where 4 is made final and transition $(3, b, 5)$ is added (Figure 9.4(b)). In the next recursive level, we call AHONEN for every orbit of A. We only consider the non-trivial orbit $\{2, 3, 4\}$ here, with its orbit automaton A_2 in Figure 9.4(c). As there are no A_2-consistent symbols and A_2 has only one orbit, we recursively merge states 2 and 4. (Line 7 gives us a choice of which transition to take, but any choice would lead us to the merging of 2 and 4.) After the merge, a is

Algorithm 6 The ForceOrbitProperty procedure.

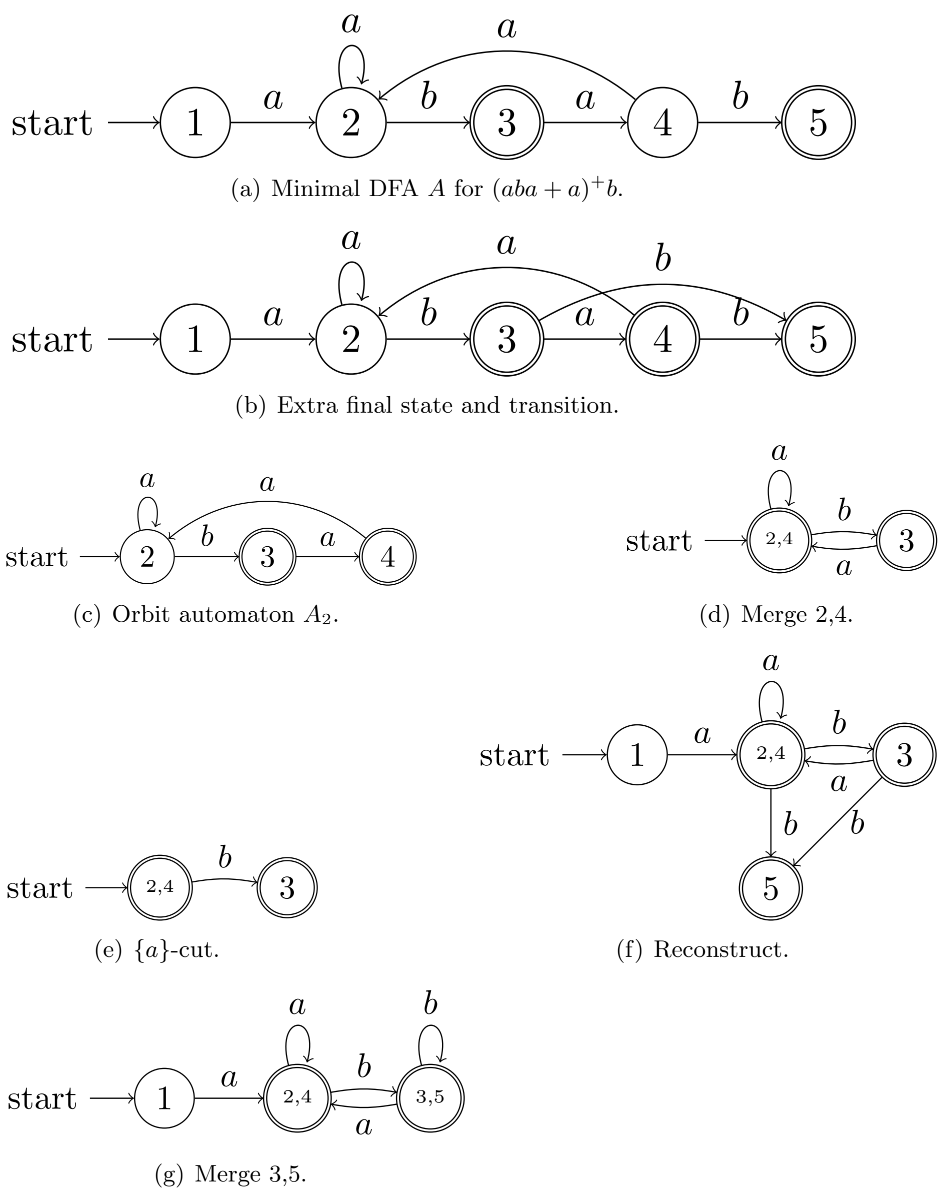
```
Require: DFA $A$
Ensure: DFA $B$ such that $L(A) \subseteq L(B)$
    for each orbit $O$ of $A$ do
      Let $g_1, \dots, g_k$ be the gates of $K$
      if there exists a gate $g_i \in F$ then
        $F \leftarrow F \cup \{g_1, \dots, g_k\}$
      for each ordered pair of gates $(g_i, g_j)$ do
        while there is an $a$ s.t. $(g_i, a, q) \in \delta$
            for $q$ outside Orbit$(g_i)$ and $(g_j, a, q) \notin \delta$ do
          Add $(g_j, a, q)$ to $\delta$
          while $(g_j, a, q') \in \delta$ for $q' \neq q$ do
            Merge$(q,q')$
```

A_2-consistent. We therefore call ForceOrbitproperty on the $\{a\}$-cut of A_2 as in Figure 9.4(e). Here, we discover that there are only two trivial orbits left, and the algorithm ends.

It remains to return from the recursion and construct the resulting DFA of the algorithm. Plugging the DFA from Figure 9.4(d) into the DFA from Figure 9.4(b) results in Figure 9.4(f). Notice that this automaton is non-deterministic. Therefore, we have to merge states 3 and 5 in order to restore determinism. The final DFA obtained by the algorithm is in Figure 9.4(g). Notice that this is a non-minimal DFA defining the language $a(a+b)^*$.

It remains to discuss Ahonen's original algorithm [3]. It is essentially the same as the one we already presented, with a slightly different ForceOrbit-Property (see Algorithm 7). We did not succeed in recovering the actual implementation of Ahonen. As the algorithm presented by Ahonen is slightly incorrect, we had to mend it in order to make a fair comparison. In particular, we observed the following: (a) The if-test on l.9 should not be there. Otherwise, the output will certainly not always be an automaton that fulfills the orbit property. (b) The if-test on l.8 should, in our opinion, be some kind of for-loop. Currently, the algorithm does not necessarily choose the same a for each pair of gates (g_i, g_j). We do believe, however, that these differences are merely typos and that Ahonen actually intended to present this new version.

Several additional remarks should be made about the original paper [3]: *(i)* it does not prove that the input DFA for Algorithm 5 is transformed into an automaton that can always be converted into a deterministic regular expression; *(ii)* it does not formally explain how states of the automaton should be merged; we have chosen the most reasonable definition; and *(iii)* it does


(a) Minimal DFA A for $(aba + a)^+b$.

(b) Extra final state and transition.

(c) Orbit automaton A_2.

(d) Merge 2,4.

(e) $\{a\}$-cut.

(f) Reconstruct.

(g) Merge 3,5.

Figure 9.4: Example run of the adapted Ahonen's algorithm.

not explain how the output DFA should be reconstructed when going back up in the recursion. Therefore, we had to make some assumptions. For example, we assume that, when re-combining orbits into a large automaton, we have a transition (q_1, a, q_2) if and only if there were subsets $q_1' \subseteq q_1$ and $q_2' \subseteq q_2$ such that the original automaton had a transition (q_1', a, q_2'). (See, for example, the

Algorithm 7 The original ForceOrbitProperty.

```
   ForceOrbitProperty($A = (Q, \Sigma, \delta, q_0, F)$)
2: for each orbit $C$ of $A$ do
     Let $g_1, \ldots, g_k$ be the gates of $K$
4:   if there exists a gate $g_i \in F$ then
       $F \leftarrow F \cup \{g_1, \ldots, g_k\}$
6:   for each ordered pair of gates $(g_i, g_j)$ do
       if there is an $a$ s.t. $(g_i, a, q) \in \delta$
8:       for $q$ outside Orbit($g_i$) and $(g_j, a, q) \notin \delta$ then
         if $(q_j, a, q') \in \delta$ for $q' \neq q$ then
10:        Add $(g_j, a, q)$ to $\delta$
           Merge $q$ and $q'$
```

transition $(\{2, 4\}, b, \{5\})$ in Figure 9.4(f), which is there because the original automaton in Figure 9.4(b) had a transition $(\{4\}, b, \{5\})$.)

With respect to remark *(i)*, we noticed in our experiments that Ahonen's algorithm sometimes indeed outputs a DFA that cannot be converted into an equivalent deterministic expression. If this happens, we reiterate Ahonen's algorithm to the thus far constructed DFA until the resulting DFA defines a deterministic language.

9.3.4 Ahonen's Algorithm Followed by Grow

Ahonen's algorithm ahonen-bkw relies on bkw to construct the corresponding deterministic expression. However, as we know, bkw generates very large expressions. We therefore consider the algorithm ahonen-grow which runs grow on the DFA resulting from ahonen.

9.3.5 Shrink

As a final approach, we present shrink. The latter algorithm operates on *state-labeled finite automata* instead of standard DFAs. Recall that for state-labeled automata, the function symbol : $Q \to \Sigma$ associates each state with its corresponding symbol. For instance, the automaton in Figure 9.4(a) is state-labeled and has symbol(2) $= a$, symbol(3) $= b$, symbol(4) $= a$, and symbol(5) $= b$; symbol(1) is undefined as 1 does not have incoming transitions.

We note that from any finite automaton, we can easily construct an equivalent state-labeled automaton by duplicating states which have more than one symbol on their incoming transitions. In particular, from a minimal DFA, one can thus construct a minimal state-labeled DFA.

Algorithm 8 The SHRINK algorithm with pool size P.

```
Require: Minimal state-labeled DFA $A$, $P \in \mathbb{N}$
Ensure: Array of det. reg. exp. $s$ with $L(A) \subseteq L(s)$
    Pool ← $\{A\}$
 2: BestArray ← empty array of $|A| - |\Sigma|$ elements
    while Pool is not empty do
 4:   $j \leftarrow 1$
      for each $B \in$ Pool do
 6:     for each pair of states $q_1, q_2$ of $B$ with symbol($q_1$) = symbol($q_2$) do
          $B_j \leftarrow$ MERGE($B$,$q_1$,$q_2$)
 8:       $j \leftarrow j + 1$
      Pool ← RANK($P$,$\{B_1, \ldots, B_{j-1}\}$)
10:   for each $B_k \in$ Pool do
        $r_{k,1} \leftarrow$ KOA-TO-KORE($B_k$)
12:     $r_{k,2} \leftarrow$ GROW($B_k$)
      for each $\ell$, $|\Sigma| \leq \ell \leq |A|$ do
14:     BestArray[$\ell$] ← the deterministic regexp $r$ of size $\ell$ from BestArray[$\ell$]
                          and all $r_{k,x}$ with maximal value proximity($A, r$)
16: return BestArray
```

The philosophy behind SHRINK rather opposes the one behind GROW: it tries to reduce the number of states of the input automaton by merging pairs of states with the same label, until every state has a different label.The result of SHRINK is an array containing deterministic expressions for which the language proximity to the target language is maximal among the deterministic expressions of the same size.

SHRINK is presented in Algorithm 8. The call to MERGE(B, q_1, q_2) is the one we explained in Section 9.3.3 and operates on the DFA B. RANK(P,$\{B_1, \ldots, B_j\}$) is a ranking procedure that selects the P "best" automata from the set $\{B_1, \ldots, B_j\}$. Thereto, we say that an automaton B_i is *better* than B_j when $L(B_i)$ is deterministic but $L(B_j)$ is nondeterministic. Otherwise, if $L(B_i)$ and $L(B_j)$ are either both deterministic or both nondeterministic, B_i is better than B_j if and only if proximity(B_i, A) > proximity(B_j, A). That is, we favour automata which define deterministic languages (as we are looking for deterministic languages), and when no distinction is made in this manner, we favour those automata which form the best approximation of the original language.

KOA-TO-KORE is an algorithm of [8] which transforms a state-labeled automaton A into a (possibly nondeterministic) expression r such that $L(A) \subseteq$

$L(r)$. Further r contains one symbol for every labeled state in A. As we are only interested in deterministic expressions, we discard the result of KOA-TO-KORE when it is nondeterministic. However, if every state of A is labeled with a different symbol, then the resulting expression also contains every symbol only once, and hence is deterministic. As SHRINK will always generate automata which have this property, SHRINK is thus guaranteed to always output at least one deterministic approximation.

Notice that SHRINK always terminates: the automata in Pool become smaller in each iteration and when they have $|\Sigma|$ states left, no more merges can be performed.

9.4 Experiments

In this section we validate our approach by means of an experimental analysis. All experiments were performed using a prototype implementation written in Java executed on a Pentium M 2 GHz with 1GB RAM. As the XML Schema specification forbids ambiguous content models, it is difficult to extract real-world expressions violating UPA from the Web. We therefore test our algorithms on a sizable and diverse set of generated regular expressions. To this end, we apply the synthetic regular expression generator used in [8] to generate 2100 nondeterministic regular expressions. From this set, 1200 define deterministic languages while the others do not. We utilize three parameters to obtain a versatile sample. The first parameter is the *size* of the expressions (number of occurrences of alphabet symbols) and ranges from 5 to 50. The second parameter is the *average number of occurrences* of alphabet symbols in the expression, denoted by κ. That is, when the size of r is n, then $\kappa(r) = n/|\text{Char}(r)|$, where $\text{Char}(r)$ is the set of different alphabet symbols occurring in r. For instance, when $r = a(a+b)^+acab$, then $\kappa(r) = 7/3 = 2.3$. In our sample, κ ranges from 1 to 5. At first glance, the maximum value of 5 for κ might seem small. However, the latter value must not be confused with the maximum number of occurrences of a single alphabet symbol, which in our sample ranges from 1 to 10. Finally, the third parameter measures how much the language of a generated expression overlaps with Σ^*, which is measured by $\text{proximity}(r, \Sigma^*)$. The expressions are generated in such a way that the parameter covers the complete spectrum uniformly from 0 to 1.

9.4.1 Deciding Determinism

As a sanity check, we first ran the algorithm BKW$^{\text{dec}}$ on the real world deterministic expressions obtained in the study [10] (which are all deterministic). On average they were decided to define a deterministic regular language within

35 milliseconds. This outcome is not very surprising as κ for each of these expressions is close to 1 (cfr. Proposition 100). We then ran the algorithm BKW$^{\text{dec}}$ on each of the 2100 expressions and were surprised that on average no more than 50 milliseconds were needed, even for the largest expressions of size 50. Upon examining these expressions more closely, we discovered that all of them have small corresponding *minimal* DFAs: on average 25 states or less. Apparently random regular expressions do not suffer much from the theoretical worst case exponential size increase when translated into DFAs.

9.4.1.1 Discussion

Although the problem of deciding determinism is theoretically intractable (Theorem 99), in practice, there does not seem to be a problem so we can safely use BKW$^{\text{dec}}$ as a basic building block of SUPAC (Algorithm 9).

9.4.2 Constructing Deterministic Regular Expressions

In this section, we compare the deterministic regular expressions generated by the three algorithms: BKW, BKW-OPT, and GROW. We point out that the comparison with BKW is not a fair one, as the latter was not defined with efficiency in mind.

Table 9.1 depicts the average sizes of the expressions generated by the three methods (again size refers to number of symbol occurrences), with the average running times in brackets. Input size refers to the size of the input regular expressions. Here, the pool-size and depth of GROW are 100 and 5, respectively. We note that for every expression individually the output of BKW is always larger than that of BKW-OPT and that GROW, when it succeeds, always gives the smallest expression. Due to the exponential nature of the BKW algorithm, both BKW and BKW-OPT can not be used for input expressions of size larger than 20.[4] For smaller input expressions, BKW-OPT is better than BKW, but still returns expressions which are in general too large to be easily interpreted. In strong contrast, when it succeeds, GROW produces very concise expressions, roughly the size of the input expression.

It remains to discuss the effectiveness of GROW. In Table 9.2, we give the success rates and the average running times for various sizes of input expressions and for several values for pool-size and depth. It is readily seen that the success rate of GROW is inversely proportional to the input size, starting at 90% for input size 5, but deteriorating to 20% for input size 25. Further, Table 9.2 also shows that increasing the pool-size or depth only has

[4]For expressions of size 20, BKW already returned expressions of size 560.000.

input size	BKW	BKW-OPT	GROW
5	9 (< 0.1)	7 (< 0.1)	3 (< 0.1)
10	216 (< 0.1)	95 (0.1)	6 (0.2)
15	1577 (0.2)	394 (0.6)	9 (0.6)
20	/	/	12 (1.5)
25-30	/	/	13 (4.0)
35-50	/	/	23 (19.6)

Table 9.1: Average output sizes and running times (in brackets, in seconds) of BKW, BKW-OPT and GROW on expressions of different input size.

input size	(d:5,p:20)	(5,100)	(10,20)	(10,100)
5	89 (< 0.1)	89 (< 0.1)	89 (< 0.1)	89 (< 0.1)
10	66 (< 0.1)	68 (0.2)	68 (0.1)	70 (0.5)
15	43 (0.1)	46 (0.6)	44 (0.3)	47 (1.6)
20	31 (0.3)	33 (1.5)	31 (0.8)	33 (3.8)
25-30	21 (0.8)	21 (4.0)	21 (1.8)	21 (9.1)
35-50	7 (3.9)	8 (19.6)	7 (8.3)	8 (43.7)

Table 9.2: Success rates (%) and average running times (in brackets, in seconds) of GROW for different values of the depth (d) and pool-size (p) parameters.

a minor impact on the success rate of GROW, but a bigger influence on its running time.

9.4.2.1 Discussion

GROW is the preferred method to run in a first try. When it gives a result it is always a concise one. Its success rate is inversely proportional to the size of the input expressions and quite reasonable for expressions up to size 20. Should GROW fail, it is not a real option to try BKW-OPT as on expressions of that size it never produces a reasonable result. In that case, the best option is to look for a concise approximation (as implemented in Algorithm 9).

9.4.3 Approximating Deterministic Regular Expressions

We now compare the algorithms AHONEN-BKW, SHRINK, and AHONEN-GROW. Note that AHONEN-BKW and AHONEN-GROW return a single approximation, whereas SHRINK returns a set of expressions (with a tradeoff between size and proximity). To simplify the discussion, we take from the output of SHRINK the expression with the best proximity, disregarding the size of the expressions.

input size	AHONEN-BKW	AHONEN-GROW	SHRINK
5	0.73 (100%)	0.71 (75%)	0.75 (100%)
10	0.81 (100%)	0.79 (56%)	0.78 (100%)
15	0.84 (100%)	0.88 (40%)	0.79 (100%)
20	/	0.89 (18%)	0.76 (100%)
25-30	/	0.89 (8%)	0.71 (100%)
35-50	/	0.75 (4%)	0.68 (100%)

Table 9.3: Quality of approximations of AHONEN-BKW, AHONEN-GROW, and SHRINK (closer to one is better). Success rates in brackets.

input size	AHONEN-BKW	AHONEN-GROW	SHRINK
5	8 (100%)	3 (75%)	3 (100%)
10	28 (100%)	6 (56%)	6 (100%)
15	73 (100%)	8 (40%)	8 (100%)
20	/	11 (18%)	10 (100%)
25-30	/	11 (8%)	13 (100%)
35-50	/	14 (4%)	18 (100%)

Table 9.4: Average output sizes of AHONEN-BKW, AHONEN-GROW, and SHRINK. Success rates in brackets.

This is justified as all expressions returned by SHRINK are concise by definition. In a practical scenario, however, the choice in tradeoff between proximity and conciseness can be left to the user.

Table 9.3 then shows the average proximity(r, s) where r is the input expression and s is the expression produced by the algorithm. As AHONEN-GROW is not guaranteed to produce an expression, the success rates are given in brackets. In contrast, AHONEN-BKW and SHRINK always return a deterministic expression. Further, as AHONEN-BKW uses BKW as a subroutine its use is restricted to input expressions of size 15.

We make several observations concerning Table 9.3. First, we see that the succes rate of AHONEN-GROW is inversely proportional to the size of the input expression. This is to be expected, as GROW is not very successful on large input expressions or automata. But, as AHONEN-BKW is also not suited for larger input expressions, only SHRINK produces results in this segment.

Concerning the quality of the approximations, we only compare AHONEN-BKW with SHRINK because AHONEN-GROW, when it succeeds, returns an expression equivalent to AHONEN-BKW which consequently possesses the same proximity. In Table 9.3, we observe that AHONEN-BKW returns on average slightly better approximations than SHRINK. Also in absolute values, AHONEN-

Algorithm 9 Supportive UPA Checker SUPAC.

```
Require: regular expression $r$
Ensure: deterministic reg. exp. $s$ with $L(r) \subseteq L(s)$
     if $r$ is deterministic then return $r$;
  2: else if $L(r)$ is deterministic then
       if GROW($r$) succeeds then return GROW($r$)
  4:   else return best from BKW-OPT(r) and SHRINK($r$)
     else return best from AHONEN-GROW(r) and SHRINK($r$)
```

BKW returns in roughly 2/3th of the cases the best approximation with respect to proximity, and SHRINK in the other 1/3th. We further observe that the quality of the approximations of SHRINK only slightly decreases but overall remains fairly good, with 0.68 for expressions of size 50.

Table 9.4 shows the average output sizes of the different algorithms. Here, we see the advantage of AHONEN-GROW over AHONEN-BKW. When an expression is returned by AHONEN-GROW, it is much more concise than (though equivalent to) the output of AHONEN-BKW. Furthermore, it has a small chance on success for those sizes of expressions on which AHONEN-BKW is not feasible anymore. Also SHRINK can be seen to always return very concise expressions.

Finally, we consider running times. On the input sizes for which AHONEN-BKW is applicable, it runs in less than a second. AHONEN-GROW was executed with pool-size 100 and depth 5 for the GROW subroutine, and took less than a second for the small input sizes (5 to 15) and up to a half a minute for the largest (50). Finally, SHRINK was executed with pool-size 10, for input expressions of size 5 to 20, and pool-size 5 for bigger expressions, and took up to a few seconds for small input expressions, and a minute on average for the largest ones.

9.4.3.1 Discussion

As running times do not pose any restriction on the applicability of the proposed methods, the best option is to always try AHONEN-GROW and SHRINK, and AHONEN-BKW only for very small input expressions (up to size 5) and subsequently pick the best expression.

9.5 SUPAC: Supportive UPA Checker

Based on the observations made in Section 9.4, we define our supportive UPA checker SUPAC as in Algorithm 9. We stress that the notion of 'best' expression can depend on both the conciseness and the proximity of the result-

input	output
$c^*cac + b$	$c^+ac + b$
$(a?bc + d)^+d$	$((a?(bc)^+)^*d^+)^+$
$((cba + c)^*b)?$	$((c^+ba?)^*b?)?$
$(c^+cb + a + c)^*$	$(a^+ + c^+b?)^*$

Table 9.5: Sample output of SUPAC.

ing regular expressions and is essentially left as a choice for the user. Note that, when GROW does not succeed, there is only a choice between a probably lengthy equivalent expressions generated by BKW-OPT or a concise approximation generated by SHRINK. In line 5, we could also make the distinction between expressions r of small size (≈ 5) and larger size (> 5). For small expressions, we then could also try AHONEN-BKW.

As an illustration, Table 9.5 lists a few small input expressions with the corresponding expression constructed by SUPAC. The first two expressions define deterministic languages, while the last two do not.

10 Conclusion

To conclude I would like to make a few observations.

First, let me point out the strong interaction between XML and formal language theory. Clearly, XML technologies are strongly rooted in formal languages. Indeed, XML schema languages are formalisms based on context-free grammars, make use of regular expressions, and thus define regular tree languages. Even the study of counting and interleaving operators present in these languages already goes back to the seventies. Therefore, XML raises questions well worth studying by the formal language theory community, such as the following:

- In Chapter 6, we discussed deterministic regular expression with counting, and showed that they can not define all regular languages. However, a good understanding of the class of languages they define is still lacking. In particular, an algorithm deciding whether a language can be defined by a weak deterministic expression with counting would be a major step forward.

- Deterministic regular expressions have the benefit of being deterministic, but mostly have negative properties. Therefore, it would be interesting for applications if another class of languages, definable by corresponding "deterministic" regular expressions, could be found that does not have these downsides. Ideally, this would include (1) an easy syntax, (2) "sufficient" expressivity, (3) good closure properties, and (4) tractable decision problems. Here, the first point is probably the most important as it is the lack of a syntax for defining all (and only) deterministic expressions, that makes them so difficult to use in practice.

The above illustrates that there is a strong link from formal language theory to XML. In this thesis, I hope to have illustrated that the opposite also holds. For instance, when studying the complexity of translating pattern-based schemas to other schema formalisms it turned out that this reduces to taking the complement and intersection of regular expressions. Studying the latter showed that this can be surprisingly difficult (double exponential), and it furthermore followed that in a translation from finite automata to regular expressions an exponential size-increase can not be avoided, even when the alphabet is fixed. These results clearly belong more to the field of formal language theory, and illustrate that XML research can add to our knowledge about formal languages.

Finally, this dual connection between XML and formal language theory can, or should, also be found in theory versus practice. The theoretical questions asked here are mostly inspired by practical problems, but it is often more difficult for theoretical results to find their way back to practice. For instance, Bruggemann-Klein and Wood's much cited paper concerning deterministic (one-unambiguous) regular languages [17] is already 10 years old, but never found its way to practical algorithms. We implemented these algorithms (see Chapter 9) and demonstrate that they can also be of practical interest. We also show in that chapter that deciding whether a language is deterministic, is PSPACE-hard but yet can be done very efficiently on all instances of interest. This is of course not a contradiction as the PSPACE-hardness only implies worst-case complexity bounds, but the term worst-case is easily forgotten in discussing the difficulty of problems. Therefore, the main lesson I learned in writing this thesis is that a fundamental study can be a very useful first step in solving a problem, but should never be its last.

11 Publications

The results presented in this thesis have been published in several papers. We list these publications here. Apart from the publications listed below, I also cooperated on a few other papers [8, 13, 33, 39].

Chapter	Reference
3	[42]
4	[37]
5	[40]
6	[38]
7	[41]
8	[40]
9	[7]

Bibliography

[1] S. Abiteboul, P. Buneman, and D. Suciu. *Data on the Web : From Relations to Semistructured Data and XML*. Morgan Kaufmann, 1999.

[2] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, 1974.

[3] H. Ahonen. Disambiguation of SGML content models. In *Principles of Document Processing (PODP 1996)*, pages 27–37, 1996.

[4] M. Almeida, N. Moreira, and R. Reis. Enumeration and generation with a string automata representation. *Theoretical Computer Science*, 387(2):93–102, 2007.

[5] M. Benedikt, W. Fan, and F. Geerts. XPath satisfiability in the presence of DTDs. *Journal of the ACM*, 55(2), 2008.

[6] P. A. Bernstein. Applying Model Management to Classical Meta Data Problems. In *Innovative Data Systems Research (CIDR 2003)*, 2003.

[7] G. J. Bex, W. Gelade, W. Martens, and F. Neven. Simplifying XML Schema: Effortless handling of nondeterministic regular expressions. In *Management of Data (SIGMOD 2009)*, 2009.

[8] G. J. Bex, W. Gelade, F. Neven, and S. Vansummeren. Learning deterministic regular expressions for the inference of schemas from XML data. In *World Wide Web Conference (WWW 2008)*, pages 825–834, 2008.

[9] G. J. Bex, F. Neven, T. Schwentick, and K. Tuyls. Inference of concise DTDs from XML data. In *Very Large Data Base (VLDB 2006)*, pages 115–126, 2006.

[10] G. J. Bex, F. Neven, and J. Van den Bussche. DTDs versus XML schema: A practical study. In *The Web and Databases (WebDB 2004)*, pages 79–84, 2004.

[11] G. J. Bex, F. Neven, and S. Vansummeren. Inferring XML Schema definitions from XML data. In *Very Large Databases (VLDB 2007)*, pages 998–1009, 2007.

[12] G. J. Bex, F. Neven, and S. Vansummeren. SchemaScope: a system for inferring and cleaning XML schemas. In *Management of Data (SIGMOD 2008)*, pages 1259–1262, 2008.

[13] H. Björklund, W. Gelade, M. Marquardt, and W. Martens. Incremental XPath evaluation. In *Database Theory (ICDT 2009)*, pages 162–173, 2009.

[14] R. Book, S. Even, S. Greibach, and G. Ott. Ambiguity in graphs and expressions. *IEEE Transactions on Computers*, 20:149–153, 1971.

[15] A. Brüggemann-Klein. Regular expressions into finite automata. *Theoretical Computer Science*, 120(2):197–213, 1993.

[16] A. Brüggemann-Klein, M. Murata, and D. Wood. Regular tree and regular hedge languages over unranked alphabets. Technical report, The Hongkong University of Science and Technologiy, April 3 2001.

[17] A. Brüggemann-Klein and D. Wood. One-unambiguous regular languages. *Information and Computation*, 142(2):182–206, 1998.

[18] J. Buchi. Weak second-order logic and finite automata. *S. Math. Logik Grundlagen Math.*, 6:66–92, 1960.

[19] A. K. Chandra, D. Kozen, and L. J. Stockmeyer. Alternation. *Journal of the ACM*, 28(1):114–133, 1981.

[20] C. Chitic and D. Rosu. On validation of XML streams using finite state machines. In *The Web and Databases (WebDB 2004)*, pages 85–90, 2004.

[21] N. Chomsky and G. A. Miller. Finite state languages. *Information and Control*, 1(2):91–112, 1958.

[22] J. Clark and M. Murata. *RELAX NG Specification.* OASIS, December 2001.

[23] R. S. Cohen. Rank-non-increasing transformations on transition graphs. *Information and Control*, 20(2):93–113, 1972.

[24] D. Colazzo, G. Ghelli, and C Sartiani. Efficient asymmetric inclusion between regular expression types. In *Database Theory (ICDT 2009)*, pages 174–182, 2009.

[25] J. Cristau, C. Löding, and W. Thomas. Deterministic automata on unranked trees. In *Fundamentals of Computation Theory (FCT 2005)*, pages 68–79, 2005.

[26] S. Dal-Zilio and D. Lugiez. XML schema, tree logic and sheaves automata. In *Rewriting Techniques and Applications (RTA 2003)*, pages 246–263, 2003.

[27] Z. R. Dang. On the complexity of a finite automaton corresponding to a generalized regular expression. *Dokl. Akad. Nauk SSSR*, 1973.

[28] A. Deutsch, M. F. Fernandez, and D. Suciu. Storing Semistructured Data with STORED. In *Management of Data (SIGMOD 1999)*, pages 431–442, 1999.

[29] L. C. Eggan. Transition graphs and the star height of regular events. *Michigan Mathematical Journal*, 10:385–397, 1963.

[30] A. Ehrenfeucht and H. Zeiger. Complexity measures for regular expressions. *Journal of Computer and System Sciences*, 12(2):134–146, 1976.

[31] K. Ellul, B. Krawetz, J. Shallit, and M. Wang. Regular expressions: New results and open problems. *Journal of Automata, Languages and Combinatorics*, 10(4):407–437, 2005.

[32] J. Esparza. Decidability and complexity of petri net problems - an introduction. In *Petri Nets*, pages 374–428, 1996.

[33] W. Fan, F. Geerts, W. Gelade, F. Neven, and A. Poggi. Complexity and composition of synthesized web services. In *Principles of Database Systems (PODS 2008)*, pages 231–240, 2008.

[34] J. Flum and M. Grohe. *Parametrized Complexity Theory.* Springer, 2006.

[35] M. Fürer. The complexity of the inequivalence problem for regular expressions with intersection. In *Automata, Languages and Programming (ICALP 1980)*, pages 234–245, 1980.

[36] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* Freeman, 1979.

[37] W. Gelade. Succinctness of regular expressions with interleaving, intersection and counting. In *Mathematical Foundations of Computer Science (MFCS 2008)*, pages 363–374, 2008.

[38] W. Gelade, M. Gyssens, and W. Martens. Regular expressions with counting: Weak versus strong determinism. In *Mathematical Foundations of Computer Science (MFCS 2009)*, 2009.

[39] W. Gelade, M. Marquardt, and T. Schwentick. The dynamic complexity of formal languages. In *Theoretical Aspects of Computer Science (STACS 2009)*, pages 481–492, 2009.

[40] W. Gelade, W. Martens, and F. Neven. Optimizing schema languages for XML: Numerical constraints and interleaving. *SIAM Journal on Computing*, 38(5):2021–2043, 2009. An extended abstract appeared in the International Conference on Database Theory (ICDT 2007).

[41] W. Gelade and F. Neven. Succinctness of pattern-based schema languages for XML. *Journal of Computer and System Sciences*. To Appear. An Extended abstract appeared in the International Symposium on Database Programming Languages (DBPL 2007).

[42] W. Gelade and F. Neven. Succinctness of the complement and intersection of regular expressions. *ACM Transactions on Computational Logic*. To appear. An extended abstract appeared in the International Symposium on Theoretical Aspects of Computer Science (STACS 2008).

[43] G. Ghelli, D. Colazzo, and C. Sartiani. Efficient inclusion for a class of XML types with interleaving and counting. In *Database Programming Languages (DBPL 2007)*, volume 4797, pages 231–245. Springer, 2007.

[44] G. Ghelli, D. Colazzo, and C. Sartiani. Linear time membership in a class of regular expressions with interleaving and counting. In *Information and Knowledge Management (CIKM 2008)*, pages 389–398, 2008.

[45] I. Glaister and J. Shallit. A lower bound technique for the size of nondeterministic finite automata. *Information Processing Letters*, 59(2):75–77, 1996.

[46] N. Globerman and D. Harel. Complexity results for two-way and multi-pebble automata and their logics. *Theoretical Computer Science*, 169(2):161–184, 1996.

[47] M. Grohe and N. Schweikardt. The succinctness of first-order logic on linear orders. *Logical Methods in Computer Science*, 1(1), 2005.

[48] H. Gruber and M. Holzer. Finite automata, digraph connectivity, and regular expression size. In *Automata, Languages and Programming (ICALP 2008)*, pages 39–50, 2008.

[49] H. Gruber and M. Holzer. Provably shorter regular expressions from deterministic finite automata. In *Developments in Language Theory (DLT 2008)*, pages 383–395, 2008.

[50] H. Gruber and M. Holzer. Language operations with regular expressions of polynomial size. *Theoretical Computer Science*, 410(35):3281–3289, 2009.

[51] H. Gruber and M. Holzer. Tight bounds on the descriptional complexity of regular expressions. In *Developments in Language Theory (DLT 2009)*, pages 276–287, 2009.

[52] H. Gruber, M. Holzer, and M. Tautschnig. Short regular expressions from finite automata: Empirical results. In *Implementation and Applications of Automata (CIAA 2009)*, pages 188–197, 2009.

[53] H. Gruber and J. Johannsen. Optimal lower bounds on regular expression size using communication complexity. In *Foundations of Software Science and Computational Structures (FOSSACS 2008)*, pages 273–286, 2008.

[54] L. Hemaspaandra and M. Ogihara. *Complexity Theory Companion.* Springer, 2002.

[55] J. E. Hopcroft, R. Motwani, and J.D. Ullman and. *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley, third edition, 2007.

[56] H. Hosoya and B. C. Pierce. Xduce: A statically typed XML processing language. *ACM Transactions on Internet Technologies*, 3(2):117–148, 2003.

[57] J. Hromkovic, S. Seibert, and T. Wilke. Translating regular expressions into small epsilon-free nondeterministic finite automata. *Journal of Computer and System Sciences*, 62(4):565–588, 2001.

[58] A. Hume. A tale of two greps. *Software, Practice and Experience*, 18(11):1063–1072, 1988.

[59] L. Ilie and S. Yu. Algorithms for computing small NFAs. In *Mathematical Foundations of Computer Science (MFCS 2002)*, pages 328–340, 2002.

[60] J. Jędrzejowicz and A. Szepietowski. Shuffle languages are in P. *Theoretical Computer Science*, 250(1-2):31–53, 2001.

[61] T. Jiang and B. Ravikumar. A note on the space complexity of some decision problems for finite automata. *Information Processing Letters*, 40(1):25–31, 1991.

[62] G. Kasneci and T. Schwentick. The complexity of reasoning about pattern-based XML schemas. In *Principles of Database Systems (PODS 2007)*, pages 155–163, 2007.

[63] P. Kilpeläinen. Inclusion of unambiguous #res is NP-hard, May 2004. Unpublished.

[64] P. Kilpeläinen and R. Tuhkanen. Regular expressions with numerical occurrence indicators — preliminary results. In *Symposium on Programming Languages and Software Tools (SPLST 2003)*, pages 163–173, 2003.

[65] P. Kilpeläinen and R. Tuhkanen. Towards efficient implementation of XML schema content models. In *Document Engineering (DOCENG 2004)*, pages 239–241, 2004.

[66] P. Kilpeläinen and R. Tuhkanen. One-unambiguity of regular expressions with numeric occurrence indicators. *Information and Computation*, 205(6):890–916, 2007.

[67] S. C. Kleene. Representation of events in nerve nets and finite automata. In *Automata Studies*, pages 3–41. Princeton University Press, 1956.

[68] C. Koch and S. Scherzinger. Attribute grammars for scalable query processing on XML streams. *VLDB Journal*, 16(3):317–342, 2007.

[69] C. Koch, S. Scherzinger, N. Schweikardt, and B. Stegmaier. Schema-based scheduling of event processors and buffer minimization for queries on structured data streams. In *Very Large Databases (VLDB 2004)*, pages 228–239, 2004.

[70] D. Kozen. Lower bounds for natural proof systems. In *Foundations of Computer Science (FOCS 1977)*, pages 254–266. IEEE, 1977.

[71] O. Kupferman and S. Zuhovitzky. An improved algorithm for the membership problem for extended regular expressions. In *Mathematical Foundations of Computer Science (MFCS 2002)*, pages 446–458, 2002.

[72] L. Libkin. Logics for unranked trees: An overview. In *Automata, Languages and Programming (ICALP 2005)*, pages 35–50, 2005.

[73] I. Manolescu, D. Florescu, and D. Kossmann. Answering XML Queries on Heterogeneous Data Sources. In *Very Large Databases (VLDB 2001)*, pages 241–250, 2001.

[74] W. Martens and F. Neven. Frontiers of tractability for typechecking simple XML transformations. *Journal of Computer and System Sciences*, 73(3):362–390, 2007.

[75] W. Martens, F. Neven, and T. Schwentick. Complexity of decision problems for simple regular expressions. In *Mathematical Foundations of Computer Science (MFCS 2004)*, pages 889–900, 2004.

[76] W. Martens, F. Neven, and T. Schwentick. Simple off the shelf abstractions for XML schema. *SIGMOD Record*, 36(3):15–22, 2007.

[77] W. Martens, F. Neven, T. Schwentick, and G. J. Bex. Expressiveness and complexity of XML schema. *ACM Transactions on Database Systems*, 31(3):770–813, 2006.

[78] W. Martens and J. Niehren. On the minimization of XML schemas and tree automata for unranked trees. *Journal of Computer and System Sciences*, 73(4):550–583, 2007.

[79] A. J. Mayer and L. J. Stockmeyer. Word problems-this time with interleaving. *Information and Computation*, 115(2):293–311, 1994.

[80] R. McNaughton. The loop complexity of pure-group events. *Information and Control*, 11(1/2):167–176, 1967.

[81] R. McNaughton. The loop complexity of regular events. *Information Sciences*, 1(3):305–328, 1969.

[82] R. McNaughton and H. Yamada. Regular expressions and state graphs for automata. *IEEE Transactions on Electronic Computers*, 9(1):39–47, 1960.

[83] A. R. Meyer and L. J. Stockmeyer. The equivalence problem for regular expressions with squaring requires exponential space. In *Switching and Automata Theory (FOCS 1972)*, pages 125–129, 1972.

[84] D. W. Mount. *Bioinformatics: Sequence and Genome Analysis.* Cold Spring Harbor Laboratory Press, September 2004.

[85] M. Murata, D. Lee, M. Mani, and K. Kawaguchi. Taxonomy of XML schema languages using formal language theory. *ACM Transactions on Internet Technologies*, 5(4):660–704, 2005.

[86] F. Neven. Automata, logic, and XML. In *Computer Science Logic (CSL 2002)*, pages 2–26, 2002.

[87] F. Neven and T. Schwentick. On the complexity of XPath containment in the presence of disjunction, DTDs, and variables. *Logical Methods in Computer Science*, 2(3), 2006.

[88] Y. Papakonstantinou and V. Vianu. DTD inference for views of XML data. In *Principles of Database Systems (PODS 2000)*, pages 35–46, New York, 2000. ACM Press.

[89] H. Petersen. Decision problems for generalized regular expressions. In *Descriptional Complexity of Automata, Grammars and Related Structures (DCAGRS 2000)*, pages 22–29, 2000.

[90] H. Petersen. The membership problem for regular expressions with intersection is complete in LOGCFL. In *Theoretical Aspects of Computer Science (STACS 2002)*, pages 513–522, 2002.

[91] G. Pighizzini and J. Shallit. Unary language operations, state complexity and Jacobsthal's function. *International Journal of Foundations of Computer Science*, 13(1):145–159, 2002.

[92] M. O. Rabin and D. Scott. Finite automata and their decision problems. *IBM Journal of Research*, 3(2):115–125, 1959.

[93] F. Reuter. An enhanced W3C XML Schema-based language binding for object oriented programming languages. Manuscript, 2006.

[94] J. M. Robson. The emptiness of complement problem for semi extended regular expressions requires c^{n} space. *Information Processing Letters*, 9(5):220–222, 1979.

[95] A. Salomaa. Two complete axiom systems for the algebra of regular events. *Journal of the ACM*, 13(1):158–169, 1966.

[96] R. Schott and J. C. Spehner. Shuffle of words and araucaria trees. *Fundamenta Informatica*, 74(4):579–601, 2006.

[97] M. P. Schutzenberger. On finite monoids having only trivial subgroups. *Information and Control*, 8(2):190–194, 1965.

[98] T. Schwentick. Automata for XML - a survey. *Journal of Computer and System Sciences*, 73(3):289–315, 2007.

[99] H. Seidl. Haskell overloading is DEXPTIME-complete. *Information Processing Letters*, 52(2):57–60, 1994.

[100] C. M. Sperberg-McQueen and H. Thompson. XML Schema. http://www.w3.org/XML/Schema, 2005.

[101] C.M. Sperberg-McQueen. Notes on finite state automata with counters. http://www.w3.org/XML/2004/05/msm-cfa.html, 2004.

[102] L. J. Stockmeyer and A. R. Meyer. Word problems requiring exponential time: Preliminary report. In *Theory of Computing (STOC 1973)*, pages 1–9. ACM Press, 1973.

[103] E. van der Vlist. *XML Schema.* O'Reilly, 2002.

[104] P. van Emde Boas. The convenience of tilings. In *Complexity, Logic and Recursion Theory*, volume 187 of *Lecture Notes in Pure and Applied Mathematics*, pages 331–363. Marcel Dekker Inc., 1997.

[105] M. Y. Vardi. From monadic logic to PSL. In *Pillars of Computer Science*, pages 656–681, 2008.

[106] V. Vianu. Logic as a query language: From frege to XML. In *Theoretical Aspects of Computer Science (STACS 2003)*, pages 1–12, 2003.

[107] V. Waizenegger. Uber die Effizienz der Darstellung durch reguläre Ausdrücke und endliche Automaten. Diplomarbeit, RWTH Aachen, 2000.

[108] L. Wall, T. Christiansen, and J. Orwant. *Programming Perl.* O'Reilly, third edition, 2000.

[109] G. Wang, M. Liu, J. X. Yu, B. Sun, G. Yu, J. Lv, and H. Lu. Effective schema-based XML query optimization techniques. In *International Database Engineering and Applications Symposium (IDEAS 2003)*, pages 230–235, 2003.

[110] S. Yu. Regular languages. In G. Rozenberg and A. Salomaa, editors, *Handbook of formal languages*, volume 1, chapter 2, pages 41–110. Springer, 1997.

[111] S. Yu. State complexity of regular languages. *Journal of Automata, Languages and Combinatorics*, 6(2):221–234, 2001.

[112] Djelloul Ziadi. Regular expression for a language without empty word. *Theoretical Computer Science*, 163(1&2):309–315, 1996.

Samenvatting

De reguliere talen vormen een zeer robuuste verzameling van talen. Zij kunnen worden voorgesteld door veel verschillende formalismen, waaronder eindige automaten, reguliere uitdrukkingen, eindige monoides, monadische tweede-orde logica, rechts-lineare grammatica's, kwantor-vrije eerste-orde updates, ... Reguliere talen zijn gesloten onder een groot aantal operaties en, nog belangrijker, bijna ieder interessant probleem met betrekking tot reguliere talen is beslisbaar. Allerhande aspecten van de reguliere talen zijn bestudeerd in de laatste 50 jaar. Vanuit een praktisch perspectief is de meest populaire manier om reguliere talen te specifiëren, gebruik te maken van reguliere uitdrukkingen. Ze worden dan ook gebruikt in toepassingen in veel verschillende gebieden van de informatica, waaronder bio-informatica, programmeertalen, automatische verificatie, en XML schema talen.

XML is de lingua franca en de de facto standaard voor het uitwisselen van data op het Web. Wanneer twee partijen data uitwisselen in XML, voldoen de XML documenten meestal aan een bepaald formaat. XML schema talen worden gebruikt om vast te leggen welke structuur een XML document mag hebben. De meest gebruikte XML schema talen zijn DTD, XML Schema, beide W3C standaarden, en Relax NG. Vanuit het oogpunt van de theorie der formele talen is ieder van deze talen een grammatica-gebaseerd formalisme met reguliere uidrukkingen aan de rechterkant. Deze uitdrukkingen verschillen echter van de standaard reguliere uitdrukkingen omdat zij zijn uitgebreid met extra operatoren, maar ook beperkt door de vereiste dat zij deterministisch moeten zijn. Hoewel deze zaken zijn opgenomen in W3C en ISO standaarden, is het niet duidelijk wat hun impact is op de verschillende schema talen. Het doel van deze thesis is daarom een studie van deze gevolgen. In het bijzonder bestuderen we de complexiteit van het optimaliseren van XML schema's in de aanwezigheid van deze operatoren, we illustreren de moeilijkheden in het migreren van een schemataal naar een andere, en we bestuderen de implicaties van de eis tot determinisme (ook in de aanwezigheid van een tel-operator), en proberen de vereiste toegankelijk te maken in de praktijk.

Hoewel de vragen die we ons stellen hoofdzakelijk geinspireerd zijn door vragen over XML, geloven we dat de antwoorden ook interessant zijn voor de algemene theoretische informatica gemeenschap. Daarom bestaat deze thesis uit twee delen. Na de inleiding en definities in de eerste twee hoofdstukken, bestuderen we in het eerste deel fundamentele aspecten van reguliere uitdrukkingen. In het tweede deel passen we deze resultaten toe op vragen met betrekking tot XML schema talen. Hoewel het grootste deel van het werk van fundamentele aard is, ontwikkelden we ook software om deze theoretische inzichten naar de praktijk over te brengen.

Fundamenten van Reguliere Uitdrukkingen

In Hoofdstuk 3 behandelen we de beknoptheid van reguliere uitdrukkingen. In het bijzonder behandelen we de volgende vragen, waarbij, zoals gewoonlijk, $L(r)$ staat voor de taal gedefinieerd door een reguliere uitdrukkingen r. Gegeven reguliere uitdrukkingen $r, r_1, \ldots, r_k$ over een alfabet Σ,

1. wat is de complexiteit van het construeren van een reguliere uitdrukking $r_\neg$ die $\Sigma^* \setminus L(r)$ definieert, i.e., het complement van r?

2. wat is de complexiteit van het construeren van een reguliere uitdrukking $r_\cap$ die $L(r_1) \cap \cdots \cap L(r_k)$ definieert?

In beide gevallen vereist het naieve algoritme dubbel exponentiële tijd in de grootte van de invoer. Inderdaad, voor het complement, vertaal r naar een NFA en determiniseer deze (eerste exponentiële stap), complementeer deze en vertaal terug naar een reguliere uitdrukking (tweede exponentiële stap). Voor intersectie is er een gelijkaardige procedure met behulp van een vertaling naar eindige automaten, het product van deze automaten te construeren, en terug te vertalen naar een reguliere uitdrukking. Merk op dat beide algoritmen niet enkel dubbel exponentiële tijd vereisen, maar in het slechtste geval ook uitdrukkingen van dubbel exponentiële grootte construeren. We tonen aan dat deze naieve algoritmen niet substantieel kunnen worden verbeterd door een klasse van reguliere uitdrukkingen te construeren waarvoor deze dubbel exponentiële vergroting van de expressies niet kan worden vermeden. De grootste technische contributie van dit hoofdstuk is een veralgemening van een resultaat van Ehrenfeucht en Zeiger. We construeren een familie van talen over een alfabet dat slechts twee symbolen bevat en tonen aan dat deze talen niet kunnen worden gedefinieerd door kleine reguliere uitdrukkingen. Hieruit volgt bovendien dat, in het slechtste geval, een vertaling van DFAs[1] naar reguliere

[1]We gebruiken in deze nederlandse samenvatting toch de engelstalige afkortingen, zoals hier van Deterministic Finite Automaton.

uitdrukkingen exponentieel is, zelfs wanneer het alfabet slechts uit 2 symbolen bestaat.

Bovendien behandelen we dezelfde vragen voor twee deelverzamelingen: enkel-voorkomende en deterministische reguliere uitdrukkingen. Een reguliere uitdrukking is enkel-voorkomend wanneer ieder alfabetsymbool slechts éénmaal voorkomt in de uitdrukking. Bijvoorbeeld, $(a + b)^*$ is een enkel-voorkomende reguliere uitdrukking (SORE) terwijl $a^*(a + b)^+$ dat niet is. Ondanks hun eenvoud omvatten SOREs toch de overgrote meerderheid van XML schema's op het Web. Determinisme vereist intuïtief dat, wanneer een woord van links naar rechts tegen een uitdrukking wordt gevalideerd, het altijd duidelijk is welk symbool in het woord overeenkomt met welk symbool in de uitdrukking. Bijvoorbeeld, de uitdrukking $(a+b)^*a$ is niet deterministisch, maar de equivalente uitdrukking $b^*a(b^*a)^*$ is het wel. De XML schema talen DTD en XML Schema vereisen dat alle uitdrukkingen deterministisch zijn. Zonder in detail te gaan, kunnen we stellen dat voor beide klassen het complementeren van uitdrukkingen makkelijker wordt, terwijl intersectie moeilijk blijft.

Waar Hoofdstuk 3 de complexiteit van het toepassen van operaties op reguliere uitdrukkingen onderzocht, bestuderen we in Hoofdstuk 4 de beknoptheid van uitgebreide reguliere uitdrukkingen, i.e. uitdrukkingen uitgebreid met extra operatoren. In het bijzonder bestuderen we de beknoptheid van uitdrukkingen uitgebreid met operatoren om te tellen (RE(#)), doorsnede te nemen (RE($\cap$)), en woorden te mengen (RE(&)). De tel-operator laat expressies zoals $a^{2,5}$ toe, die uitdrukt dat er minstens 2 en ten hoogste 5 a's moeten voorkomen. Deze RE(#)s worden gebruikt in egrep en Perl uitdrukkingen en XML Schema. De klasse RE($\cap$) is een goed bestudeerde extensie van de reguliere uitdrukkingen en wordt ook vaak de half-uitgebreide reguliere uitdrukkingen genoemd. De meng-operator laat uitdrukkingen zoals $a \& b \& c$ toe, die specifieert dat a, b, en c in eender welke volgorde mogen voorkomen. Deze operator komt voor in de XML schema taal Relax NG [22] en, in een zeer beperkte vorm, XML Schema. We geven een volledig overzicht van de beknoptheid van deze klassen van uitgebreide reguliere uitdrukkingen met betrekking tot standaard reguliere uitdrukkingen, NFAs, en DFAs.

De belangrijkste reden om de complexiteit van een vertaling van uitgebreide reguliere uitdrukkingen naar standaard reguliere uitdrukkingen te bestuderen, is om meer inzicht te krijgen in de kracht van de verschillende operatoren. Deze resultaten hebben echter ook belangrijke gevolgen met betrekking tot de complexiteit van het vertalen van een schemataal naar een andere. Zo tonen we bijvoorbeeld aan dat in een vertaling van RE(&) naar standaard RE een dubbel exponentiële vergroting in het algemeen niet vermeden kan worden. Aangezien Relax NG de meng-operator toelaat, en XML Schema deze alleen in een zeer beperkte vorm toelaat, impliceert dit dat ook in een vertaling van Relax NG naar XML Schema zo een dubbel exponentiële vergroting niet te vermijden is. Desondanks, aangezien XML Schema een wijdverspreide W3C standaard is, en Relax NG een meer flexibel alternatief is, zou zo een vertaling meer dan wenselijk zijn. Ten tweede bestuderen we de beknoptheid van reguliere uitdrukkingen ten opzichte van eindige automaten aangezien, wanneer algoritmische problemen worden beschouwd, de gemakkelijkste oplossing vaak via een vertaling naar automaten gaat. Onze negatieve resultaten tonen echter aan dat het vaak een beter idee is om gespecialiseerde algoritmen, die zulke vertalingen vermijden, voor deze uitgebreide uitdrukkingen te ontwikkelen.

In Hoofdstuk 5 bestuderen we de complexiteit van het equivalentie, inclusie, en intersectie niet-leegheid probleem voor verschillende klassen van reguliere uitdrukkingen. Deze klassen zijn algemene reguliere uitdrukkingen, uitgebreid met tel- en meng-operatoren, en ketting reguliere uitdrukkingen (CHAREs), uitgebreid met de tel-operator. Deze CHAREs vormen een andere eenvoudige subklasse van de reguliere uitdrukkingen. De grootste motivatie voor het bestuderen van deze klassen ligt in het feit dat ze gebruikt worden in XML schema talen, en we in Hoofdstuk 8 de complexiteit van dezelfde problemen voor deze schema talen zullen bestuderen, gebruik makend van de resultaten in dit hoofdstuk.

In Hoofdstuk 6 bestuderen we deterministische reguliere uitdrukkingen uitgebreid met de tel-operator, aangezien dit essentieel de reguliere uitdrukkingen zijn die voorkomen in XML Schema. De meest gebruikte notie van determinisme is zoals we ze eerder definieerden: een expressie is deterministisch wanneer het bij het valideren van een woord steeds duidelijk is welk symbool in het woord overeenkomt met welk symbool in de uitdrukking. We kunnen echter bijkomend vereisen dat het ook steeds duidelijk is *hoe* er van één positie naar de volgende in de uitdrukking moet gegaan worden. De eerste notie zullen we zwak determinisme noemen, en de tweede sterk determinisme. Bijvoorbeeld, $(a^*)^*$ is zwak deterministisch, maar niet sterk deterministisch, aangezien het niet duidelijk is over welke ster we moeten itereren bij het gaan van één a naar de volgende.

Voor standaard reguliere uitdrukkingen wordt het verschil tussen deze twee noties zelden opgemerkt, aangezien ze voor deze uitdrukkingen bijna overeenkomen. Dit kunnen we zeggen omdat iedere zwak deterministische uitdrukking in lineaire tijd naar een sterk deterministische uitdrukking kan worden vertaald. Deze situatie verandert echter volledig wanneer de tel-operator wordt toegevoegd. Ten eerste, het algoritme om te beslissen of een uitdrukking al dan niet deterministisch is, is allesbehalve eenvoudig. Bijvoorbeeld, $(a^{2,3}+b)^{2,2}b$ is zwak determintisch terwijl $(a^{2,3}+b)^{3,3}b$ dat niet is. Ten tweede, zoals we zullen aantonen, zijn zwak deterministische uitdrukkingen met tel-operator strikt expressiever dan sterk deterministische uitdrukkingen met tel-operator. Daarom is het doel van dit hoofdstuk een studie van de noties van zwak en sterk determinisme in het bijzijn van de tel-operator met betrekking tot expressiviteit, beknoptheid, en complexiteit.

Toepassingen voor XML Schema Talen

In Hoofdstuk 7 bestuderen we patroon-gebaseerde schema talen. Dit zijn schema talen equivalent in expressieve kracht met enkel-type EDTDs, de vaak gebruikte abstractie van XML Schema. Een voordeel van deze taal is dat ze de expressieve kracht van XSDs duidelijker maakt: het inhoud model van een element hangt enkel af van reguliere woord-eigenschappen gevormd door de voorouders van dat element. Patroon-gebaseerde schema's kunnen daarom gebruikt worden als een type-vrije front-end voor XML Schema. Aangezien ze zowel in een existentiele als in een universele semantiek kunnen geïnterpreteerd worden, bestuderen we in dit hoofdstuk de complexiteit van vertalingen tussen de twee semantieken en vertalingen naar de formalismen DTDs, EDTDs, en enkel-type EDTDs, de vaak gebruikte abstracties voor respectievelijk DTD, Relax NG en XML Schema.

Hiervoor maken we gebruik van de resultaten in Hoofdstuk 3 en tonen aan dat in het algemeen vertalingen van patroon-gebaseerde schema talen naar de andere formalismen, exponentiële of zelfs dubbel exponentiële tijd vragen. Bijgevolg is er weinig hoop om patroon-gebaseerde schema's, in hun meest algemene vorm, als een nuttige front-end te gebruiken voor XML Schema. Daarom bestuderen we ook meer beperkte klassen van schema's: lineaire en sterk lineaire patroon-gebaseerde schema's. De meest gerestricteerde klasse, sterk lineaire patroon-gebaseerde schema's, laten efficiente vertalingen naar andere talen en efficiente algoritmes voor beslissingsproblemen toe, en zijn bovendien expressief genoeg om de overgrote meerderheid van XSDs voorkomend op het Web uit te drukken. Vanuit een praktisch oogpunt is dit dus een zeer interessante klasse van schema's.

In Hoofdstuk 8 bestuderen we de impact van het toevoegen van tel- en meng-operator aan reguliere uitdrukkingen voor de verschillende schema talen. We beschouwen het equivalentie-, inclusie-, en intersectie niet-leegheid probleem aangezien deze de bouwstenen voor algoritmen voor het optimaliseren van XML schema's zijn. We beschouwen ook het simplificatie probleem: Gegeven een EDTD, is deze equivalent met een enkel-type EDTD of een DTD?

Ten slotte, in Hoofdstuk 9, geven we algoritmen waarmee je kan omgaan met de vereiste in DTD en XML dat alle reguliere uitdrukkingen deterministisch moeten zijn, i.e. de UPA vereiste in XML Schema. In de meeste boeken over XML Schema wordt de UPA vereiste uitgelegd met behulp van een voorbeeld, in plaats van een duidelijke syntactische definitie. Wanneer bijgevolg, na het opstellen van het schema, één of meerdere expressies worden geweigerd omdat ze niet-deterministisch zijn, is het zeer moeilijk voor gebruikers, die geen expert zijn om de reden van de fout te vinden, en nog moeilijker om deze te corrigeren. Het doel van dit hoofdstuk is om methodes te onderzoeken om niet-deterministische uitdrukkingen te vertalen naar kleine en leesbare deterministische uitdrukkingen die ofwel dezelfde taal definiëren, ofwel een goede benadering zijn. We stellen het SUPAC algoritme voor dat kan gebruikt worden in een slimme XSD tester. Naast fouten te vinden in XSDs die niet conform UPA zijn, kan het bovendien ook goede alternatieven suggereren. Op deze manier is de gebruiker ontlast van de vervelende UPA vereiste, en kan hij zich concentreren op het ontwerpen van een juist schema.