Optimization of XML schema languages: complexity of counting and the shuffle operator

Wouter GELADE

promotor :
Prof. dr. Frank NEVEN

Universiteit Maastricht

universiteit hasselt

Eindverhandeling voorgedragen tot het bekomen van de graad
Master in de informatica afstudeerrichting databases

Acknowledgments

I would like to thank a number of people who have made this thesis possible.

First of all, special thanks go to my promotor, Prof. Dr. Frank Neven. He introduced me to the interesting problems discussed in this work. During the past year, he guided me through this thesis with ideas, advice, and feedback. For every question I had, he made time for me. I also want to thank Dr. Wim Martens for some interesting discussions on a few problems in this work.

More personally, I would like to thank my parents for giving me the opportunity to study, and for supporting me throughout my whole life and education. Finally, I thank my friends for the moments of humor and relaxation during the creation of this thesis.

Contents

1 Introduction **5**

2 Computational complexity **15**
2.1 The Turing machine 15
2.2 Time- and space complexity 18
2.3 Non-deterministic Turing machines 19
2.4 Alternating Turing machines 19
2.5 Complexity classes 20
2.6 Reductions and completeness 23

3 Tiling problems **24**

4 Regular expressions **34**
4.1 Regular expressions 34
4.2 Automata for regular expressions 36
4.3 Properties of extended NFAs 46

5 Tree languages **52**
5.1 Trees 52
5.2 Tree automata 53
5.2.1 Binary tree automata 53
5.2.2 Unranked tree automata 55
5.2.3 Equivalence of tree automata 56
5.3 XML schema languages 60
5.4 Decision problems 66

6 Complexity of regular expressions **68**
6.1 Regular expressions 68
6.1.1 Equivalence and inclusion 69
6.1.2 Intersection 73
6.2 Chain regular expressions 75

7 Complexity of unranked tree automata 85
7.1 Equivalence and inclusion 86
7.2 Intersection 100

8 Conclusion 104

Bibliography 107

Samenvatting (Dutch Summary) 110

Chapter 1

Introduction

XML (eXtensible Markup Language) ([BPSM$^+$04]) is a W3C standard for exchanging structured documents and data over the internet. It is a format that is very flexible, because it allows user-defined *tags*, and the content of an XML document is easy to interpret by both humans and computers. Therefore, XML has become the standard format for data exchange over the world wide web. An example XML document is shown in Figure 1.1.

XML schemas allow users to define their own format of XML documents. An XML schema describes the tags that can be used in the XML document, and the structure the document must have. Using XML schemas has many benefits. An example is the problem of data integration: Suppose there are two data-sources on the internet whose data is stored using XML documents. Furthermore, each data-source has an XML schema which defines all its XML documents. Then, it is possible to integrate these data-sources in one database by only studying their XML schemas, and not every XML document separately.

The most used and widespread XML schema languages are *Document Type Definitions* (DTDs) ([BPSM$^+$04]), *XML Schema* ([SMT05]), and *Relax NG* ([CM01]). The first popular XML schema language was DTD. It is a simple language that uses grammar rules with regular expressions on the right-hand sides, to describe the structure of XML documents. An example DTD which defines the XML document of Figure 1.1 is shown in Figure 1.2.

DTDs have become the standard XML schema language, but its possibilities are rather limited. Therefore, a number of new XML schema languages have been created. The most popular of these languages is XML Schema. XML Schema has an XML based syntax. So, an *XML Schema Definition* (XSD) is itself an XML document. XML Schema also has a typing system, and a lot of other useful features, which are not included in DTD. One of these is the possibility to express that a number of elements can occur in any

```
<cd>
    <song>
        <title>Susan's house</title>
        <length>223</length>
        <singlesSold>55000</singlesSold>
    </song>
    <song>
        <title>Beautiful freak</title>
        <length>213</length>
        <singlesSold>100000</singlesSold>
    </song>
    <song>
        <title>Flower</title>
        <length>227</length>
    </song>
</cd>
```

Figure 1.1: An example XML document that describes a part of the album 'Beautiful Freak' by Eels. For every song, the title and the length (in seconds) is given. If a song has been a single, the number of singles sold are described by *singlesSold*. The values are chosen randomly.

```
<!DOCTYPE cd [
  <!ELEMENT cd (song*)>
  <!ELEMENT song (title,length,singlesSold?)>
  <!ELEMENT title (#PCDATA)>
  <!ELEMENT length (#PCDATA)>
  <!ELEMENT singlesSold (#PCDATA)>
]>
```

Figure 1.2: A DTD which defines the XML document of Figure 1.1.

order. This can be done by surrounding these elements by the *all* tag. If we want to do this in a DTD, we have to list all possible permutations of the elements (n! possibilities for n children). Another feature is the possibility to indicate the minimum and maximum number an element can occur. The *minoccurs* and *maxoccurs* attributes of an element allow us to specify these numbers. Again, we can do this in a DTD, but we have to list that element *maxoccurs* times. An example XSD, which defines the XML document of Figure 1.1, is shown in Figure 1.3. Here, we use the *all* tag to specify that the elements *title*, *length*, and *singlesSold* can occur in any order. The *minoccurs* and *maxoccurs* attributes are used to indicate that a cd must contain at least one and at most twenty songs.

Another extension of DTD is Relax NG. This is a very elegant and powerful XML schema language, but has not become as popular as XML Schema. Relax NG has an XML based syntax, and a simpler equivalent *compact* syntax. Like XML Schema, it has a typing system for its elements. As an alternative for the *all* tag of XML Schema, it allows the &-operator in its regular expressions. This binary operator is called the shuffle or interleave operator, and allows the words accepted by its two operands to be shuffled. Relax NG does not have an alternative for the *minoccurs* and *maxoccurs* attributes. An example of a (part of) a Relax NG document which describes the element *song* as it is defined by the XSD in Figure 1.3, is shown in Figure 1.4. The shuffle operator is used to replace the *all* tag of XML Schema.

In this work, we are interested in the optimization of XML schema languages. An example of an optimization problem is the minimization of XML schemas. A minimized schema allows us to do document validation more efficiently, and improves the running time of other tests on the schema. To minimize a schema, we can create a smaller schema and check if it still defines the same set of XML documents as the original schema. The problem of checking whether two schemas define the same set of XML documents is the equivalence problem.

Analogously to the equivalence problem, we also define the inclusion and intersection (non-emptiness) problem.

- Equivalence: Given two schemas D_1 and D_2, do D_1 and D_2 define exactly the same set of XML documents?

- Inclusion: Given two schemas D_1 and D_2, is every document defined by D_1 also defined by D_2?

- Intersection (non-emptiness): Given schemas $D_1, \ldots, D_n$, does there exist an XML document that is defined by every D_i, $i = 1, \ldots, n$?

```
<schema>
    <element name="cd" type="cdType">

    <xs:element name="single">
        <xs:complexType>
            <xs:sequence>
                <xs:all>
                    <xs:element name="title" type="xs:string"/>
                    <xs:element name="length" type="xs:integer"/>
                    <xs:element name="singlesSold" type="xs:integer"/>
                </xs:all>
            </xs:sequence>
        </xs:complexType>
    </xs:element>

    <xs:element name="regular">
        <xs:complexType>
            <xs:sequence>
                <xs:all>
                    <xs:element name="title" type="xs:string"/>
                    <xs:element name="length" type="xs:integer"/>
                </xs:all>
            </xs:sequence>
        </xs:complexType>
    </xs:element>

    <xs:element name="cdType">
        <xs:complexType>
            <xs:sequence>
                <xs:choice minoccurs = "1" maxoccurs = "20">
                    <xs:element name="song" type="single"/>
                    <xs:element name="song" type="regular"/>
                </xs:choice>
            </xs:sequence>
        </xs:complexType>
    </xs:element>
</schema>
```

Figure 1.3: An XSD which defines the XML document in Figure 1.1, but violates the EDC constraint.

```
element song {
    element title { text }
    & element length { xsd:integer }
    & element singlesSold { xsd:integer }?
}
```

Figure 1.4: A description of the song element by a Relax NG document, in compact syntax, as defined by the XSD in Figure 1.3.

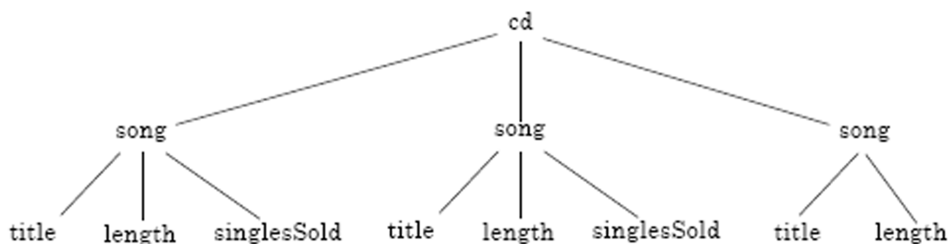

Figure 1.5: The tree representation of the XML document of Figure 1.1.

These three problems are the building blocks of most optimization problems for XML schemas. Therefore, it is important to determine the complexity of these problems for the different types of XML schema languages.

To be able to study the complexity of these problems, we have to make an abstraction of XML documents and XML schema languages. XML documents are represented by trees. The tree representation of the XML document of Figure 1.1 is shown in Figure 1.5.

We abstract from DTDs with extended context free grammars, which we will also call DTDs. These extended context free grammars use regular expressions to describe their content.

Example 1.1. *We can abstract from the DTD of Figure 1.2 by the following DTD:*

$$\begin{array}{rcl} cd & \rightarrow & song^* \\ song & \rightarrow & title\ length\ singlesSold? \end{array}$$

One of the major differences between DTD on one hand, and XML Schema and Relax NG on the other hand, is the typing system. Therefore, we add a typing system to DTDs, which gives us *extended* DTDs (EDTDs)

([PV00]). These EDTDs exactly represent Relax NG schemas. However, XML Schema enforces constraints which Relax NG does not have. Therefore the class of EDTDs is too powerful to represent XML Schema. One of these constraints is *element declarations consistent* (EDC). EDC says that in a regular expression, no elements with the same name, but with a different type can be used. We can limit the power of EDTDs to single-type EDTDs (EDTD^{st}), which exactly describe XML Schema languages with the EDC constraint ([BMNS05],[MLMK05]). These EDTD^{st}s do not allow that two elements with the same name, but a different type occur in the same expression. We use EDTD^{st}s as an abstraction for XML Schema.

Example 1.2. *We can abstract from the XSD of Figure 1.3 by the following EDTD:*

$$\begin{aligned} cd &\rightarrow (song^1 + song^2)^{1..20} \\ song^1 &\rightarrow title \;\&\; length \;\&\; singlesSold \\ song^2 &\rightarrow title \;\&\; length \end{aligned}$$

Here, $song^1$ represents the single *type songs, and $song^2$ represents the* regular *type songs. Note that this EDTD is not single-type since $song^1$ and $song^2$ occur in the same expression. This is because the XSD of Figure 1.3 does not satisfy the EDC constraint. We can solve this by rewriting our XSD.* □

It should be noted that DTD and XML Schema also require its content to be deterministic or one-unambiguously. This is expressed by the *Deterministic Content Models* constraint for DTD and the *Unique Particle Attribution* rule for XML Schema. We do not enforce these constraints in our model for DTD (DTD) and XML Schema ($EDTD^{st}$) because there has been a lot of discussion over these rules in the XML community (see, for example, page 98 of [vdV02], [Man01], and [SM03]). Therefore, it is interesting to see what the complexity of the decision problems is for DTD and XML Schema without these constraints.

DTD, XML Schema, and Relax NG, and their abstractions, all use (subclasses of) regular expressions to describe the structure of XML documents. In these regular expressions, a number of different operators are used. DTDs use the standard set of operators: concatenation ($\cdot$), Kleene-star (*), and disjunction (+). XML Schema adds a number of features. As we have seen, the *all* tag can be translated using the shuffle operator (&). The *mincount* and *maxcount* attributes can be translated using the numerical occurrence operator. The use of this numerical occurrence operator is shown in Example 1.2, and indicates that a cd contains 1 to 20 songs. We denote this operator

by #. By specifying the operators that are allowed in the regular expressions, we can define subsets of the regular expressions.

It turns out that the complexity of the decision problems (equivalence, inclusion, and intersection) for DTDs, and EDTD^{st} is highly correlated with the complexity of the same decision problems for the subclass of regular expressions they use ([MNS04]). We also define unranked tree automata (UTA), which use (subclasses of) regular expressions too. These unranked tree automata are equally powerful as EDTDs. Therefore, the complexity of the decision problems for unranked tree automata and EDTDs is the same. Because of this close correspondence between DTDs, EDTDs, and EDTD^{st}s on one hand, and regular expressions and unranked tree automata on the other hand, we will study the complexity of the decision problems for various subclasses of regular expressions and unranked tree automata.

It is well known that any traditional $\text{RE}(+, \cdot, *)$-expression can be translated into an equivalent non-deterministic finite automaton (NFA) of polynomial size. These NFAs can then be used to determine upper bounds on the decision problems for $\text{RE}(+, \cdot, *)$. However, translating an $\text{RE}(+, \cdot, *, \&, \#)$-expression directly into an equivalent NFA gives a double-exponential blowup. Therefore, we introduce a new kind of automata, *extended* NFAs (ENFAs), which extend the capabilities of the regular NFAs, and allow us to translate any $\text{RE}(+, \cdot, *, \&, \#)$-expression directly into such an ENFA of polynomial size. We often use these ENFAs to determine upper bounds for classes using the numerical occurrence or the shuffle operator.

We first consider a number of subclasses of regular expressions, which are defined by allowing a number of operators. For example, the class $\text{RE}(+, \cdot, *)$ defines all regular expressions which only use the $+$, $\cdot$, and $*$ operators. Some of the complexities were already known in the literature, and are summarized in Table 1.1. The new results obtained in this work are summarized in Table 1.2.

	equivalence	inclusion	intersection
$\text{RE}(+, \cdot, *)$	PSPACE [SM73]	PSPACE [SM73]	PSPACE [Koz77]
$\text{RE}(+, \cdot, *, \&)$	EXPSPACE [MS94]	EXPSPACE [MS94]	

Table 1.1: Complexity classes for problems with regular expression which were already known in the literature. All complexities are completeness results.

These subsets of regular expressions all contain very complex expressions. However, the regular expressions used in practical DTDs and XSDs are mostly very simple. Bex, Neven, and Van den Bussche have done a study

	equivalence	inclusion	intersection
RE($+, \cdot, *, \#$)	EXPSPACE	EXPSPACE	PSPACE
RE($+, \cdot, *, \&$)			PSPACE
RE($+, \cdot, *, \#, \&$)	EXPSPACE	EXPSPACE	PSPACE

Table 1.2: Complexity classes for problems with regular expressions, found in this work. All complexities are completeness results.

on this subject ([BNVdB04]), and it turns out that more than ninety percent of the regular expressions occurring in practical DTDs and XSDs are expressions $e_1 \cdots e_n$, where every e_i is a factor of the form $(x_1 + \cdots + x_n)$, possibly extended with the *, +, or ? operators, and each x_i is a string. Martens, Neven, and Schwentick, have characterized this class of regular expressions as CHain Regular Expressions (CHAREs), and studied the complexity of the decision problems for a number of subclasses of these CHAREs ([MNS04]). We extend these CHAREs with the numerical occurrence operator. We present some of the results of Martens et. al. in Table 1.3. The results obtained in this work are summarized in Table 1.4.

	equivalence	inclusion	intersection
CHARE($S \backslash \{\#\}$)	in PSPACE	PSPACE	PSPACE
CHARE($a, a?$)	in PTIME	coNP	NP
CHARE($a, a*$)	in PTIME	coNP	NP

Table 1.3: Complexity classes for a selection of problems with CHAREs due to Martens et. al ([MNS04]). Here, CHARE($S \backslash \{\#\}$) denotes the full subset of CHAREs as defined by Martens et. al, and CHARE($a, a?$) (resp. CHARE($a, a*$)) are the simple subsets of CHAREs where only factors of the form a or $a?$ (resp. $a*$), $a \in \Sigma$, are allowed. All complexities are completeness results, unless mention otherwise.

Finally, we investigate the complexity of the decision problems for UTAs. The complexity of these problems depends on the subset of regular expressions used in the transition function of the UTA. For example, the class UTA(RE($+, \cdot, *, \&$)) contains all UTAs which only use the three standard operators and the shuffle operator in their transition functions. The only result known in the literature is that the equivalence problem and intersection problem for UTA(RE($+, \cdot, *$)) are EXPTIME-complete ([Sei90], [Sei94]). The results found in this work are summarized in Table 1.5.

	equivalence	inclusion	intersection
CHARE(S)	in EXPSPACE	EXPSPACE	PSPACE
CHARE($a, a?, a\#$)	in PTIME	coNP-hard, in EXPSPACE	NP
CHARE($a, a\#^{>0}$)	in PTIME	in PTIME	in PTIME

Table 1.4: Complexity classes for problems with CHAREs, found in this work. Here, CHARE(S) denotes the full subset of CHAREs extended with the numerical occurrence operator, and CHARE($a, a?, a\#$) is the simple subset of CHAREs where only factors of the form a, $a?$, and $a^{k..l}$ are allowed. The class CHARE($a, a\#^{>0}$) is equivalent to CHARE($a, a?, a\#$), with the difference that we do not allow factors of the form $a?$, and that the lower bound of the numerical occurrence operator, k, must be bigger than zero. All complexities are completeness results, unless mention otherwise.

	equivalence	inclusion	intersection
UTA(RE($+, \cdot, *$))		EXPTIME	
UTA(RE($+, \cdot, *, \#$))	EXPSPACE	EXPSPACE	EXPTIME
UTA(RE($+, \cdot, *, \&$))	EXPSPACE	EXPSPACE	EXPTIME
UTA(RE($+, \cdot, *, \#, \&$))	EXPSPACE	EXPSPACE	EXPTIME

Table 1.5: Complexity classes for problems with unranked tree automata, found in this work. All complexities are completeness results, unless mentioned otherwise.

Using these results and the earlier mentioned correlation between the complexities of the decision problems for regular expressions, and unranked tree automata on one hand, and DTDs, EDTDs, and EDTDst on the other hand, we can establish the complexity of the decision problem for DTDs, EDTDs, and EDTDst. Since these classes are abstractions of DTD, Relax NG, and XML Schema, this gives us the complexity of the decision problems for DTD, Relax NG, and XML Schema.

In chapter 2, we define a number of aspects of the complexity theory that are important for this work. We first define Turing machines (TM) and time- and space complexity, based on these Turing machines. We also define two more powerful Turing machines, non-deterministic TMs and alternating TMs. Next, we describe the complexity classes we consider in this work and formally define the notions reduction and completeness.

In chapter 3, we consider tiling problems. This is a class of problems that

has complete problems for many interesting complexity classes. The corridor tiling and the 2-player corridor tiling problem are PSPACE and EXPTIME complete. We introduce the 2^n-corridor tiling problem and show that it is EXPSPACE-complete. These tiling problems are used to show hardness in a number of proofs in chapters 6 and 7.

In chapter 4, we first define regular expressions and chain regular expressions. Then, we introduce NFAs and extended NFAs, and show how a $\mathrm{RE}(+, \cdot, *, \#, \&)$-expression can be converted into an equivalent ENFA. Finally, we give some complexity results concerning ENFAs.

In chapter 5, we consider tree languages. We first describe how we will abstract from XML documents by trees. To recognize these trees, we need tree automata. We define a number of tree automata, and for these tree automata we show which have equal power and which are less powerful than the other. Then, we give the definitions of DTDs, EDTDs, and EDTD^{st}s and formally define the decision problems we consider. Finally, we give the correspondence between the complexity of the decision problems for regular expression and unranked tree automata on one hand, and DTDs, EDTDs, and EDTD^{st}s on the other hand.

In chapter 6, we investigate the complexity of the equivalence, inclusion and intersection problem for regular expressions. First, we do this for the full class of regular expressions. Here, we investigate what the addition of the numerical occurrence and shuffle operator does for the complexity of our decision problems. Then, we consider much simpler subclasses of regular expressions, CHain Regular Expressions (CHAREs).

In chapter 7, we investigate the complexity of the decision problems for unranked tree automata. For tree automata we can also consider automata that only use a subset of the classes of regular expressions.

In chapter 8, we present some conclusions about the results of this work.

Chapter 2

Computational complexity

We first define some basic terms. In the rest of this thesis, Σ always denotes a finite alphabet. A Σ-*symbol* (or simply symbol) is an element of Σ, and a Σ-*string* (or simply string) is a finite sequence $x = a_1 \cdots a_n$ of Σ-symbols. We define the length of x, denoted by $|x|$, to be n. We denote the empty string by ε. The set of *positions of x* is $\{1, \ldots, n\}$ and the *symbol of x at position i* is a_i. By $x_1 \cdot x_2$ we denote the *concatenation* of two strings x_1 and x_2. For readability, we sometimes also denote the concatenation of x_1 and x_2 by $x_1 x_2$.

Most definitions in this chapter are based on the book *computational complexity* by Papadimitriou ([Pap94]).

2.1 The Turing machine

The k-tape Turing machine (TM) is a mathematical model of a computer. A k-tape TM consists of k tapes. Each tape has a head that can be used to read and write the tape. The tapes are unbounded to the right and consist of cells which contain symbols. Furthermore, a TM is always in a state and has a transition function. Based on this transition function, the current state and the symbols the heads of the tapes are currently reading, the TM can write a symbol to each of its tapes and/or move its heads. We will use the k-tape TM as our formal model of computation. The TM is formally defined as follows:

Definition 2.1. *A k-tape* Turing machine*, where k is an integer, is a quadruple, $M = (Q, \Sigma, \delta, q_0)$, and where*

1. *Q is the finite set of states;*

2. *Σ is the finite alphabet, and Σ always contains the special blank symbol $\sqcup$ and the special start symbol $\rhd$;*

3. *$\delta : Q \times \Sigma^k \to (Q \cup \{q_{halt}, q_{accept}, q_{reject}\}) \times (\Sigma \times \{\to, \leftarrow, _\})^k$ is the transition function; and*

4. *$q_0 \in Q$ is the start state.*

We assume that q_{halt} (the halting state), q_{accept} (the accepting state), q_{reject} (the rejecting state), and the head directions $\leftarrow$ for "left", $\to$ for "right", and $_$ for "stay", are not in $Q \cup \Sigma$.

The transition function δ controls the actions the TM takes. Intuitively, $\delta(q, a_1, \ldots, a_k) = (p, b_1, D_1, \ldots, b_k, D_k)$ means that, if M is in state q, the head of the first string is scanning a a_1, that of the second a a_2, and so on, then the next state will be p, the first head will write b_1 and move in the direction indicated by D_1, and so on for the other heads.

We put a few constraints on δ to restrict the use of the special start symbol $\rhd$. When a head reads the $\rhd$, the head has to move right and the $\rhd$ has to be rewritten. This makes sure that the tape heads will never fall of the left end of a tape: If $a_i = \rhd$, then $b_i = \rhd$ and $D_i = \to$ must hold. The $\rhd$ can not be written at any other place but the first position of a tape: If $b_i = \rhd$, then $a_i = \rhd$ must hold. Since this transition function is a function and not a relation, we call this TM a *deterministic*.

The computation begins by writing the special start symbol $\rhd$ followed by the input x on the first tape. The other tapes only contain the start symbol $\rhd$. The tape heads point to the first position of their tape, the $\rhd$. The current state is initialized as q_0, the start state.

From here on, the transition function will step by step change the current state, move the heads and write to the tapes. When the head of a tape is above the last symbol of that tape and has to move to the right, we write a $\sqcup$ at the end of that tape. Thus we ensure that the tapes are truly unbound to the right.

Since the heads can never fall of the left hand side of the tape and the tapes are unbounded to the right, there is only one reason why the TM will halt. That occurs when it reaches a halting state: q_{halt}, q_{accept}, or q_{reject}.

To describe the computation of a TM formally we need a way to describe the complete state the computation is in, a *configuration*. A configuration of M is a $(2k+1)$-tuple $(q, w_1, u_1, \ldots, w_k, u_k)$, where $q \in Q$ is a state and w_i, u_i are strings in Σ^*. Here, q is the current state; w_i is the string to the left of the head on the ith tape, including the symbol scanned by the head; and u_i is the string to the right of the head on the ith tape, possibly empty.

We say that configuration $(q, w_1, u_1, \dots, w_k, u_k)$ *yields in one step* configuration $(q, w'_1, u'_1, \dots, w'_k, u'_k)$, denoted $(q, w_1, u_1, \dots, w_k, u_k) \rightarrow^M (q, w'_1, u'_1, \dots, w'_k, u'_k)$ if the following is true. First suppose that a_i is the last symbol of w_i, for $i = 1, \dots, k$. We have the following: If $D_i = \rightarrow$, then w'_i is w_i with its last symbol (which was a_i) replaced by b_i, and the first symbol of u_i appended to it ($\sqcup$ if u_i is the empty string); u'_i is u_i with the first symbol removed (or, if u_i was the empty string, u'_i remains empty). If $D_i = \leftarrow$, then w'_i is w_i with a_i omitted from its end, and u'_i is u_i with b_i attached in the beginning. Finally, if $D_i = _$, then w'_i is w_i with the ending a_i replaced by b_i, and $u'_i = u_i$.

Based on the relationship *yields in one step*, we can now define *yields* as its transitive closure. We say that configuration $(q, w_1, u_1, \dots, w_k, u_k)$ *yields in l steps* configuration $(q, w'_1, u'_1, \dots, w'_k, u'_k)$, denoted $(q, w_1, u_1, \dots, w_k, u_k) \rightarrow^{M^l} (q, w'_1, u'_1, \dots, w'_k, u'_k)$, where $k \in \mathbb{N}$, if there exist configurations $(q, w_1^i, u_1^i, \dots, w_k^i, u_k^i), i = 1, \cdots, l+1$, such that $(q, w_1^i, u_1^i, \dots, w_k^i, u_k^i) \rightarrow^M (q, w_1^{i+1}, u_1^{i+1}, \dots, w_k^{i+1}, u_k^{i+1})$ for $i = 1, \dots, l$, $(q, w_1, u_1, \dots, w_k, u_k) = (q, w_1^1, u_1^1, \dots, w_k^1, u_k^1)$ and $(q, w'_1, u'_1, \dots, w'_k, u'_k) = (q, w_1^{l+1}, u_1^{l+1}, \dots, w_k^{l+1}, u_k^{l+1})$. Finally, we say that configuration $(q, w_1, u_1, \dots, w_k, u_k)$ *yields* configuration $(q, w'_1, u'_1, \dots, w'_k, u'_k)$, denoted $(q, w_1, u_1, \dots, w_k, u_k) \rightarrow^{M^*} (q, w'_1, u'_1, \dots, w'_k, u'_k)$, if there is a $l \geq 0$ such that $(q, w_1, u_1, \dots, w_k, u_k) \rightarrow^{M^l} (q, w'_1, u'_1, \dots, w'_k, u'_k)$.

Definition 2.2. *Let M be a TM and x the input string for M;*

- M accepts x *if* $(q_0, \rhd, x, \rhd, \varepsilon, \dots, \rhd, \varepsilon) \rightarrow^{M^*} (q_{accept}, u_1, w_1, \dots, u_k, w_k)$, *for some $u_i, w_i \in \Sigma^*$, $i = 1, \dots k$. We say that $M(x) =$ "accept".*

- M rejects x *if* $(q_0, \rhd, x, \rhd, \varepsilon, \dots, \rhd, \varepsilon) \rightarrow^{M^*} (q_{reject}, u_1, w_1, \dots, u_k, w_k)$, *for some $u_i, w_i \in \Sigma^*$, $i = 1, \dots k$. We then say that $M(x) =$ "reject".*

- *If $(q_0, \rhd, x, \rhd, \varepsilon, \dots, \rhd, \varepsilon) \rightarrow^{M^*} (q_{halt}, u_1, w_1, \dots, u_k, w_k)$, for some u_i, $w_i \in \Sigma^*$, $i = 1, \dots k$, we say that $M(x) = u_k w_k$. That is, M calculates a function from strings to strings and its output can be found on the last tape.*

- *If M does not end on input x, we say that $M(x) = \nearrow$.*

Definition 2.3. *Let $L \subset (\Sigma \backslash \{\sqcup, \rhd\})^*$ be a language, that is, a set of strings of symbols.*

- *Let M be a TM such that for every $x \in (\Sigma \backslash \{\sqcup, \rhd\})^*$ holds that if $x \in L$, then $M(x) =$ "accept", and if $x \notin L$, then $M(x) =$ "reject". We then say that M decides L. If L is decided by a TM M, then L is called a* recursive *language.*

- *We say that a TM M* accepts L *when for every $x \in (\Sigma - \{\llcorner, \triangleright\})^*$ holds that if $x \in L$, then $M(x) =$ "accept" and if $x \notin L$, then $M(x) = \nearrow$. If L is accepted by some TM M, we say that L is* recursively enumerable.
- *For a TM M, define $L(M) = \{x \in \Sigma^* \mid M(x) = \text{"accept"}\}$ as the language accepted by M.*

2.2 Time- and space complexity

We will only consider Turing machines that stop on every input. We can classify these Turing machines, and the problems they solve, based on the time and space they use.

Let M be a TM that takes as input x and say $(q_0, \triangleright, x, \triangleright, \varepsilon, \ldots, \triangleright, \varepsilon) \rightarrow^{M^t} (H, u_1, w_1, \ldots, u_k, w_k)$, for $H \in \{q_{accept}, q_{reject}\}$, and $u_i, w_i \in \Sigma^*$, $i = 1, \ldots, k$.

Definition 2.4. *The* time required by M on input *x is t. Suppose that a language $L \subset (\Sigma \backslash \{\llcorner, \triangleright\})^*$ is decided by a TM M that for every $x \in (\Sigma \backslash \{\llcorner, \triangleright\})^*$ requires no more than $f(|x|)$ time, $f : \mathbb{N} \rightarrow \mathbb{N}$ being a non-decreasing function. We then say that $L \in TIME(f(n))$, and $f(n)$ is a* time bound *for M. The class $TIME(f(n))$ is the collection of Turing machines that operate in no more than $f(n)$ time.*

Definition 2.5. *The* space required by M on input x *is $max\{|u_i| + |w_i| \mid i = 1, \ldots, k\}$, the maximum number of tape cells used by the longest tape during the computation. Suppose that a language $L \subset (\Sigma \backslash \{\llcorner, \triangleright\})^*$ is decided by a TM M that for every $x \in (\Sigma - \{\llcorner, \triangleright\})^*$ requires no more than $f(|x|)$ space, $f : \mathbb{N} \rightarrow \mathbb{N}$ being a non-decreasing function. We then say that $L \in SPACE(f(n))$, and $f(n)$ is a* space bound *for M. The class $SPACE(f(n))$ is the collection of Turing machines that operate in no more than $f(n)$ space.*

The number of tapes used by a TM is not important when we consider its complexity.

Theorem 2.6. *Given any k-tape Turing machine M operating within time $f(n)$, there exists a one-tape Turing machine M' operating within time $O(f(n)^2)$ such that, for any input x, $M(x) = M(x')$.*

The following Theorems show that multiplicative constants are not important.

Theorem 2.7. *Let $L \in TIME(f(n))$. Then, for any $\varepsilon > 0$, $L \in TIME(f'(n))$, where $f'(n) = \varepsilon f(n) + n + 2$.*

Theorem 2.8. *Let $L \in SPACE(f(n))$. Then, for any $\varepsilon > 0$, $L \in SPACE(f'(n))$, where $f'(n) = 2 + \varepsilon f(n)$.*

2.3 Non-deterministic Turing machines

Definition 2.9. *A* non-deterministic Turing machine (NTM) *is a deterministic Turing machine where the transition function is of the form* $\delta : Q \times \Sigma^k \to P((Q \cup \{q_{halt}, q_{accept}, q_{reject}\}) \times (\Sigma \times \{\to, \leftarrow, _\})^k)$*, where* P *stands for powerset. We thus allow the Turing machine to make non-deterministic choices in every step. We say that a NTM* M decides *a language* $L \subset (\Sigma - \{\sqcup, \rhd\})^*$ *if for every* $x \in (\Sigma \backslash \{\sqcup, \rhd\})^*$ *holds that* $x \in L$ *if and only if there exist* $u_i, w_i \in \Sigma$*,* $i = 1, \ldots, k$*, such that* $(q_0, \rhd, x, \rhd, \varepsilon, \ldots, \rhd, \varepsilon) \to^{M^*} (q_{accept}, u_1, w_1, \ldots, u_k, w_k)$*.*

We say that a NTM M *decides language* L *in time* $f(n)$*, if* N *decides* L*, and, moreover for any* $x \in \Sigma^*$*, if* $(q_0, \rhd, x, \rhd, \varepsilon, \ldots, \rhd, \varepsilon) \to^{M^k} (q, u_1, w_1, \ldots, u_k, w_k)$*, then* $k \leq f(|x|)$*. That is, we require that* M*, besides deciding* L*, does not have computation paths longer than* $f(n)$*, where* n *is the length of the input. The set of languages decided by NTMs within time* f *is denoted by NTIME(*$f(n)$*). The class NSPACE(*$f(n)$*) is defined analogously for non-deterministic space complexity. Here, we do not allow any computation path to use more than* $f(n)$ *space, where* n *is the length of the input.*

2.4 Alternating Turing machines

A non-deterministic Turing machine increases the power of a deterministic Turing machine. Similarly, an alternating Turing machine will extend the power of the non-deterministic Turing machine. First, let us take a different definition of the non-deterministic Turing machine: A configuration *leads to acceptance* if and only if it is either a final accepting configuration or (recursively) at least one of its successors leads to acceptance. That is, each configuration is in some sense an implicit OR of its successor configuration.

Suppose now that we also allow configurations to be AND configurations and thus only lead to acceptance if all its successors lead to acceptance. We will say that a configuration is in the *mode* AND or OR. The mode of each configuration is determined by the state of the configuration. The state set is thus divided in two distinct sets, the AND (or universal) states and the OR (or existential) states. The machine accepts its input if and only if the initial configuration leads to acceptance. The alternating Turing machine is formally defined as follows:

Definition 2.10. *An* alternating Turing machine (ATM) *is a non deterministic Turing machine* $M = (Q, \Sigma, \delta, q_0)$ *in which the set of states* Q *is partitioned into two sets,* $Q = Q_{AND} \uplus Q_{OR}$*. Let* x *be an input and consider the tree of computations of* M *on input* x*. Each node in this tree is a configuration of the precise machine and includes the step number of the*

machine. The children of a node with configuration γ in this tree are all configurations γ' such that γ yields in one step γ'. The leaf nodes are configurations which contain a halting state. Define now recursively, starting from the leaves of the tree and going up, a subset of these configurations, called the eventually accepting configurations *as follows: First, all leaf configurations with state q_{accept} are eventually accepting. A configuration γ with state in Q_{AND} is eventually accepting if and only if all of its successor configurations (configurations γ' such that γ yields in one step γ') are eventually accepting. A configuration γ with state in Q_{OR} is eventually accepting if and only if at least one of its successor configurations is eventually accepting.*

We say that M accepts *x if the initial configuration is eventually accepting. We say that an ATM M* decides *a language L if M accepts all strings $x \in L$ and rejects all strings $x \notin L$. We let ATIME($f(n)$) be the class of all languages decided by an alternating Turing machine, for which all computations on input x halt after at most $f(|x|)$ steps. Analogously, ASPACE($f(n)$) is the class of all languages decided by an alternating Turing machine that uses no more than $f(|x|)$ space on input x.*

2.5 Complexity classes

Languages and their corresponding Turing machines can be classified based on their time and space requirements. A complexity class is a collection of languages that satisfy certain space or time conditions. The Turing machines that decide these languages can be deterministic, non-deterministic or alternating Turing machines. The complexity classes used in this work are:

- P = PTIME = $\bigcup_{k\geq 1}$ TIME(n^k);
- NP = NPTIME = $\bigcup_{k\geq 1}$ NTIME(n^k);
- EXPTIME = $\bigcup_{k\geq 1}$ TIME(2^{n^k});
- NEXPTIME = $\bigcup_{k\geq 1}$ NTIME(2^{n^k});
- L = LOGSPACE = SPACE($\log n$);
- NL = NLOGSPACE = NSPACE($\log n$);
- AL = ALOGSPACE = ASPACE($\log n$);
- PSPACE = $\bigcup_{k\geq 1}$ SPACE(n^k);
- NPSPACE = $\bigcup_{k\geq 1}$ NSPACE(n^k);

- APSPACE = $\bigcup_{k \geq 1}$ ASPACE(n^k);
- EXPSPACE = $\bigcup_{k \geq 1}$ SPACE(2^{n^k}); and
- NEXPSPACE = $\bigcup_{k \geq 1}$ NSPACE(2^{n^k}).

These complexity classes have been widely studied. We give a few important results concerning their relationships. First, we must define the class of functions we consider.

Definition 2.11. *Let $f : \mathbb{N} \to \mathbb{N}$. We say that f is a* proper complexity function *if f is nondecreasing (that is, $f(n+1) \geq f(n)$ for all n), and furthermore the following is true: There is a k-string Turing machine M such that, for any integer n, and any input x of length n, $(q_0, \rhd, x, \rhd, \varepsilon, \ldots, \rhd, \varepsilon) \to^{M^t} (h, x, \rhd, \sqcup^{j_2}, \ldots, \rhd, \sqcup^{j_3}, \ldots, \rhd, \sqcup^{j_n}, \rhd, \sqcap^{f(x)})$, such that $t = O(n + f(n))$, and the $j_i = O(f(|x|))$ for $i = 2, \ldots, k-1$, with t and the j_i's depending only on n. In other words, on input x, M computes the string $\sqcap^{f(x)}$, where $\sqcap$ is a "quasi-blank" symbol. And, on any input x, M_f halts after $O(|x| + f(|x|))$ steps and uses $O(f(|x|))$ space besides its input.*

Almost every function used in practice satisfies this definition. Examples of proper complexity functions are $\log n^2, n \log n, n^2, n^3 + 3n, 2^n, \sqrt{n}$, and $n!$.

Theorem 2.12. *([Pap94]) Suppose that a language L is decided by a nondeterministic Turing machine N in time $f(n)$. Then it is decided by a 3-tape deterministic Turing machine N in time $O(c^{f(n)})$, where $c > 1$ is some constant depending on N.*

Corollary 2.13.

$$NTIME(f(n)) \subseteq \bigcup_{c>1} TIME(c^{f(n)}) \tag{2.1}$$

and thus,

$$NP \subseteq EXPTIME \tag{2.2}$$

Theorem 2.14. *(The time hierarchy Theorem) If $f(n) \geq n$ is a proper complexity function, then the class TIME($f(n)$) is properly contained within TIME($f(2n+1)^3$).*

Corollary 2.15.

$$P \subset EXPTIME \tag{2.3}$$

Theorem 2.16. *(The space hierarchy Theorem) If $f(n)$ is a proper complexity function, then the class SPACE($f(n)$) is properly contained within SPACE($f(2n+1)^3$).*

Corollary 2.17.

$$L \subset PSPACE \tag{2.4}$$
$$PSPACE \subset EXPSPACE \tag{2.5}$$

Theorem 2.18. *([Pap94])*

$$SPACE(f(n)) \subseteq NSPACE(f(n)) \tag{2.6}$$
$$NSPACE(f(n)) \subseteq ASPACE(f(n)) \tag{2.7}$$
$$TIME(f(n)) \subseteq NTIME(f(n)) \tag{2.8}$$
$$NTIME(f(n)) \subseteq SPACE(f(n)) \tag{2.9}$$
$$NSPACE(f(n)) \subseteq TIME(k^{\log n + f(n)}) \tag{2.10}$$

Theorem 2.19. *(Savitch) If $f(n) \geq \log n$ is a proper complexity function, then $NSPACE(f(n)) \subseteq SPACE(f^2(n))$.*

Corollary 2.20.

$$PSPACE = NPSPACE \tag{2.11}$$
$$EXPSPACE = NEXPSPACE \tag{2.12}$$

Theorem 2.21. *([CKS81])*

$$ASPACE(f(n)) = TIME(k^{f(n)}) \tag{2.13}$$

Corollary 2.22.

$$AL = P \tag{2.14}$$
$$APSPACE = EXPTIME \tag{2.15}$$

Using these theorems, we can put together the following equation:
L $\subseteq$ NL $\subseteq$ AL = P $\subseteq$ NP $\subseteq$ PSPACE = NPSPACE $\subseteq$ APSPACE = EXPTIME $\subseteq$ NEXPTIME $\subseteq$ EXPSPACE = NEXPSPACE

Here, we already know that L $\neq$ PSPACE and PSPACE $\neq$ EXPSPACE. It follows that at least one of the inclusions between L, NL, P, NP and PSPACE must be strict. The same holds for the inclusions between PSPACE, EXPTIME, NEXPTIME and EXPSPACE. It is generally assumed, but not proven, that all these inclusions are strict.

Definition 2.23. *Let $L \subseteq (\Sigma \backslash \{\sqcup, \triangleright\})^*$ be a language. The complement of L is defined as $\overline{L} = (\Sigma \backslash \{\sqcup, \triangleright\})^* \backslash L$. If C is a complexity class, then $coC = \{\overline{L} \mid L \in C\}$.*

The classes L, NL, P, EXPTIME, AL, PSPACE, NPSPACE, and APSPACE are closed under complement. NP is closed under complement if and only if P = NP, NEXPTIME if and only if EXPTIME = NEXPTIME.

2.6 Reductions and completeness

Definition 2.24. *We say that language L_1 is* logspace reducible*, or simply reducible, to language L_2 if there is a function F from strings to strings computable by a deterministic TM M in LOGSPACE such that for all inputs x the following is true: $x \in L_1$ if and only if $F(x) \in L_2$. The function F is called a* reduction *from L_1 to L_2.*

Turing machines in LOGSPACE have a read-only input tape and a read-write output tape. Only operations on the output tape count when computing the complexity of the TM. In this way we allow the TM to read its entire input. This extra feature does not increase the power of the TM.

Definition 2.25. *Let C be a complexity class, and let L be a language in C. We say that L is* C-complete *if any language $L' \in C$ can be reduced to L.*

It is important to establish complete problems for a complexity class. These problems represent the class since they possess the essence and difficulty of that class.

Definition 2.26. *We say that a class C' is* closed under reductions *if, whenever L is reducible to L' and $L' \in C'$, then also $L \in C'$*

Most interesting complexity classes, including every class defined in the previous section, are closed under reduction. Furthermore, we know that if two classes C and C' are both closed under reductions, and there is a language L which is complete for both C and C', then $C = C'$.

Chapter 3

Tiling problems

Definition 3.1. *In a* tiling problem *we are given a set of tile types O. A* tile *is a square of unit size which is divided in four triangles by the two diagonals. A* tile type *o is obtained by selecting a color from a finite set of colors for each of the four triangles, denoted $o = < l, t, r, b >$, where l, t, r, and b represent the left, top, right, and bottom color respectively. Tile types can not be rotated or reflected.*

A tiling *is a mapping from a subset of the square grid in the plane to a set of tile types O, $til : \{1, \dots, m\} \times \{1, \dots, n\} \rightarrow O$ for $n, m \in \mathbb{N}$. That is, a covering of a region in the plane by instances of the tiles in O. Furthermore, this mapping must have the property that two tiles which share a horizontal or vertical edge in the plane have equal colors for the two triangles adjacent to this edge. We also say that O* tiles *the $m \times n$-corridor.*

In this context we define the sets H and V, which represent the horizontal and vertical constraints. Let $o_i = < l_i, t_i, r_i, b_i >$ and $o_j = < l_j, t_j, r_j, b_j >$ then $H = \{(o_i, o_j) \mid o_i, o_j \in O \text{ and } r_i = l_j\}$ and $V = \{(o_i, o_j) \mid o_i, o_j \in O \text{ and } t_i = b_j\}$. An alternative definition of a tiling is a mapping til for which $(til(k, l), til(k + 1, l)) \in V$, for $1 \leq k < m$ and $1 \leq l \leq n$; and $(til(k, l), til(k, l + 1)) \in H$, for $1 \leq k \leq m$ and $1 \leq l < n$.

Definition 3.2. *In the* corridor tiling problem, *we are given an instance $\tau = \langle O, \overline{b}, \overline{t}, n \rangle$, where O is a finite set of tile types, $\overline{b}$, $\overline{t}$ are n-tuples of tiles and $n \in \mathbb{N}$ is given in unary. Here, $\overline{b}$, and $\overline{t}$ stand for the bottom and top row, respectively. We have to decide whether there exists a valid tiling $til : \{1, \dots, m\} \times \{1, \dots, n\} \rightarrow O$, for some $m \in \mathbb{N}$, such that $\overline{b} = (til(1, 1), \dots, til(1, n))$, and $\overline{t} = (til(m, 1), \dots, til(m, n))$. Such a tiling is a* corridor tiling.

Definition 3.3. *The* 2^n-corridor tiling problem *is the problem: given an instance $\tau = \langle O, o_{bot}, o_{top}, n \rangle$, where O is a finite set of tile types, $o_{bot}, o_{top} \in O$*

are tile types and $n \in \mathbb{N}$ is given in unary, decide whether O tiles the $m \times 2^n$-corridor for some $m \in \mathbb{N}$ in such a way that o_{bot} is placed at $(1, 1)$ and o_{top} is placed at $(m, 1)$. Such a tiling is a 2^n-corridor tiling.

We can extend tiling problems to tiling games. In such a game, we have two players: CONSTRUCTOR and SPOILER. These players can place one tile on turn. CONSTRUCTOR tries to create a valid tiling, and SPOILER tries to prevent that. CONSTRUCTOR wins the game if SPOILER places an illegal tile (a tile whose borders don't match the borders of the tiles that were already placed), or if a valid tiling is constructed. We say that CONSTRUCTOR has a *winning strategy* if he wins no matter what SPOILER does.

Definition 3.4. *The* 2-player corridor tiling problem*, is the problem: Given an instance $\tau = \langle O, \bar{b}, \bar{t}, n \rangle$ for the corridor tiling problem, does CONSTRUCTOR have a winning strategy for the corridor tiling game on τ.*

Theorem 3.5.

- *The corridor tiling problem is PSPACE-complete.*
- *The 2-player corridor tiling problem is EXPTIME-complete.*
- *The 2^n-corridor tiling problem is EXPSPACE-complete.*

Proof. The first and second result are due to Chlebus ([Chl86]). We show that the 2^n-corridor tiling problem is in EXPSPACE (Lemma 3.6), and that it is EXPSPACE-hard (Lemma 3.7). □

Lemma 3.6. *The 2^n-corridor tiling problem is in EXPSPACE.*

Proof. We give a non-deterministic exponential space algorithm that given an input $\tau = \langle O, o_{bot}, o_{top}, n \rangle$ for the 2^n-corridor tiling problem, decides whether there exists a 2^n-corridor tiling for τ. Since NEXPSPACE = EXPSPACE, this shows that the problem is in EXPSPACE.

The following algorithm always stores two consecutive rows of tiles, *currentRow* and *previousRow*.

1. Start by guessing the start row and call it *currentRow*. The first tile is o_{bot} and all other tiles are guessed.
2. Check the horizontal constraints in the *currentRow*. If they are not satisfied, REJECT. If they are satisfied and if the first tile of the *currentRow* is o_{top}, ACCEPT.
3. Let *previousRow* = *currentRow* and guess a new *currentRow*. Check the vertical constraints between the *previousRow* and the *currentRow*. If they are not satisfied, REJECT.

4. GOTO step 2.

This algorithm tries to guess a correct tiling for the $m \times 2^n$-corridor. It starts by guessing the bottom row and checks if it is a valid row. Then, it keeps on guessing new rows on top of the previous one. For each new row it checks for each of its tiles if it is compatible with its neighbors. If it is a valid row and if its first tile is o_{top}, we have constructed a correct 2^n-corridor tiling for τ.

If there exists a tiling for τ, there is a run of the algorithm that constructs that tiling and accepts. If there does not exist such a tiling, the TM will never accept.

It only remains to show that the computation is done using only exponential space. We always store 2 rows of tiles of length 2^n. Since n is given in unary in the input, this requires exponential space. We have to check the vertical and horizontal constraints of the tiles in steps 2 and 3. This can be done by running through the set of tiles and thus only requires logarithmic space pointers to the input.

□

Lemma 3.7. *The 2^n-corridor tiling problem is EXPSPACE-hard.*

Proof. To show that the 2^n-corridor tiling problem is EXPSPACE-hard, we have to prove that every language $L \in EXPSPACE$ can be reduced to the 2^n-corridor tiling problem, using a log-space reduction. We have to find a function F, such that for every string x, $F(x)$ is the encoding of a set of tile-types O, an integer $n > 0$ and two tiles $o_{bot}, o_{top} \in O$, so that there exists a 2^n-corridor tiling for $\langle O, o_{bot}, o_{top}, n\rangle$ if and only if $x \in L$.

The only property of L that we know is that it is in EXPSPACE. So, there must exist a Turing machine $M = (Q, \Sigma, \delta, q_0)$ that decides L and that on input x never uses more than $2^{(|x|)^k}$ space, for some constant k. To be able to do the reduction, we first have to create a Turing machine M' that also decides L but differs in a few aspects to M. We call the Turing machine that does the reduction R.

Proposition 3.8. *For every exponential space Turing machine $M = (Q, \Sigma, \delta, q_0)$ and string $x = a_1 \cdots a_n$, $a_i \in \Sigma$, we can construct, in logarithmic space, an exponential space Turing machine $M' = (Q', \Sigma, \delta', q_0')$, such that*

1. *x is accepted by M if and only if x is accepted by M'; and*
2. *M' only accepts when the tape head is above the first tape cell, and*
3. *M' starts its computation by writing $\lhd$ on the $2^{(|x|)^k} + 1$th position of the tape.*

Proof. We describe the changes we have to make to M to generate M'. For item 2, we have to create a new accept state, q'_{accept}. We change the transition function such that when it reaches q_{accept}, it moves the head to the first position of the tape. There, it enters q'_{accept}.

For item 3, we change the transition function in such a way that when the algorithm starts, we write a $\lhd$-symbol to the $2^{(|x|)^k}+1$th position of the tape and move the tape head back to the first position of the tape.

Note that all these changes do not affect the work of M. After we have put the $\lhd$ into place, the tape head does not move over the $\lhd$ anymore. We know that M never uses more than $2^{(|x|)^k}$ space. Therefore, it only uses the first to $2^{(|x|)^k}$ positions on the tape of M', and will it never move over the $\lhd$ symbol, which is located at the $2^{(|x|)^k}+1$th position of the tape. Finally, M' reaches q'_{accept} if and only if M reaches q_{accept}. Therefore, x is accepted by M if and only if x is accepted by M'.

Our proposition requires that the changes are made in logarithmic space. For the requirement in item 2, we have to create a new accept state, and we have to add transitions that move the tape head to the first position of the tape. This can be done by a constant number of transitions, and thus in logarithmic space.

The second adjustment is the hardest. Since this reduction has to be done in logarithmic space, we can not just create $2^{(|x|)^k}+1$-states to move to the $2^{(|x|)^k}+1$th position, write the $\lhd$ and move back. Instead we start by writing $2^{(|x|)^k}$ in binary on the tape immediately after the input x and use a special token # to separate the input and the binary representation of $2^{(|x|)^k}$. Immediately after this string we write the $\lhd$-token. Our tape now looks like $\rhd\ a_1\cdots a_n\#10\cdots 0\ \lhd$ and the head is placed at the rightmost position of the binary number.

We create states that repeat the following:

1. If the number on the tape is equal to $|x|+(|x|)^k+2$, stop.
2. Decrease the number with 1.
3. Move the tape head to the $\lhd$-token, replace it with a ␣ and write an $\lhd$ on the next position to the right.

This algorithm moves the $\lhd$-token one place to the right in every iteration. Since the $\lhd$-token starts at position $|x|+(|x|)^k+3$, it will be in position $2^{(|x|)^k}+1$ after $2^{(|x|)^k}-(|x|+(|x|)^k+2)$ iterations. That is why it stops when the number on the tape is equal to $|x|+(|x|)^k+2$.

Of course, the states that execute this algorithm again have to be generated in logarithmic space. We begin by computing $(|x|)^k$ on the worktape of

R. We do this by always keeping k, $|x|$ and a temporary solution in binary on the worktape. The temporary solution is 1 in the beginning. Then, we multiply the temporary solution k times by $|x|$. We use the k as counter, diminish it by 1 after every multiplication and stop when k equals 0. At the end, the temporary solution is equal to $(|x|)^k$. Note that since $(|x|)^k$ and $|x|$ are polynomials in $|x|$, they only require logarithmic space to store. Furthermore, k is a constant and thus imposes no problem. The computation of the multiplications can also be done in logarithmic space.

Now we can create $(|x|)^k$ on our worktape, we can generate the necessary states. We first have to write $2^{(|x|)^k}$ in binary on the tape. This is a number of the form $100 \cdots 00$, with $(|x|)^k$ zeros. So, we first create a state that writes the "1". Then we create $(|x|)^k$ states, using $(|x|)^k$ as a counter, that all write a "0" and then pass control to the next state.

During the computation we have to check if the number on the tape of M' is equal to $|x| + (|x|)^k + 2$. To create the states that check that, we first generate $|x| + (|x|)^k + 2$ in binary on our worktape (of R). To be equal, the binary pattern of the number on the tape has to be the same as that of $|x| + (|x|)^k + 2$, possibly preceded by a number of zeros. So, we run from right to left through our number and check for every bit of the number if it is equal to the corresponding bit of $|x| + (|x|)^k + 2$. We can do this by creating $\log \lceil |x| + (|x|)^k + 2 \rceil$ states. If the binary pattern of the number is equal to that of $|x| + (|x|)^k + 2$, we only have to check if all the following characters, upto the #-token, are "0". If this is the case, the number is equal to $|x| + (|x|)^k + 2$. To generate this series of states, we have to run one time from right to left through the number $|x| + (|x|)^k + 2$, which we have generated on our worktape. Since $|x| + (|x|)^k + 2$ can be stored using logarithmic space, this all only requires logarithmic space.

All other operations only require a constant number of states. To decrease a number with 1, we only have to find the rightmost "1"-token (one state), change it into a "0" (one state) and change all zeros to the right of that place into a "1" (one state). After moving the tape head to the $\lhd$-token (one state), we move the $\lhd$-token one place (one state) and move the head back to the rightmost position of the number (one state).

Finally, when this algorithm is done we overwrite the number and the #-token with $\sqcup$-tokens (one state) and move the head to the first position of the tape (one state).

□

By proposition 3.8, we know that given an input x, we can transform M to an equivalent TM M' with certain properties. We will use M' in the rest of this proof. We can start by describing all input elements of the 2^n-

corridor tiling problem. These are defined in such a way that when a tiling is possible, M' (and thus M) accepts the string $x = a_1 a_2 \cdots a_{|x|}$. In that tiling, every row represents a configuration of M'. More specifically, the horizontal sides of the tiles (bottom and top) describe the tape contents, the position of the head, and the current state. The vertical sides (left and right) represent the movement of the tape head and the state of the Turing machine. The tile-types are defined as follows:

1. $\forall a \in \Sigma$: $< _, a, _, a >$;
2. $\forall q, a : \delta'(q, a) = (r, b, _)$, $r \in Q', b \in \Sigma$: $< _, (r, b), _, (q, a) >$;
3. $\forall q, a : \delta'(q, a) = (r, b, \rightarrow)$, $r \in Q', b \in \Sigma$: $< _, b, \rightarrow r, (q, a) >$;
4. $\forall a \in \Sigma, \forall q \in Q', a \neq \triangleright$: $<\rightarrow q, (q, a), _, a >$;
5. $\forall q, a : \delta'(q, a) = (r, b, \leftarrow)$, $r \in Q', b \in \Sigma$: $<\leftarrow r, b, _, (q, a) >$;
6. $\forall a \in \Sigma, \forall q \in Q', a \neq \triangleleft$: $< _, (q, a), \leftarrow q, a >$;
7. $< _, (q_0, a_1), 1, _ >$, $< |x| + 1, \sqcup, |x| + 1, _ >$ en $\forall i, 1 < i \leq |x|$: $< i, a_i, i + 1, _ >$; and
8. $< _, (q'_{accept}, \triangleright), _, (q'_{accept}, \triangleright) >$.

We define o_{bot} as $< _, (q_0, a_1), 1, _ >$ and o_{top} as $< _, (q'_{accept}, \triangleright), _, (q'_{accept}, \triangleright) >$.

We see that the horizontal sides of the tiles hold two different types of symbols, a symbol $a \in \Sigma$ or a tuple (q, a), with $q \in Q'$ and $a \in \Sigma$. A symbol a means that there is an a at that position of the tape. The tuple (q, a) means that there is an a at that position of the tape, the tape head is above that tape position and the machine is in state q. The vertical sides hold symbols of the form $_$, $\rightarrow q$ or $\leftarrow q$, with $q \in Q'$. A $_$ means that the tape head is not at that position. A $\rightarrow q$ (or $\leftarrow q$) means that the tape head is at that position, moves to the right (or left) and that the state of M' becomes q. Figure 3.1 shows the tile types again in a graphical way.

We will now discuss the different tile types. The tiles of item 1 make sure that all symbols on the tape that are not affected by the tape head, stay the same. The tiles in items 2 to 6 take care of the changes made by the head. When the tape head stays at the same position, we have to adjust the symbol at that position and the state (item 2). When the head moves to the

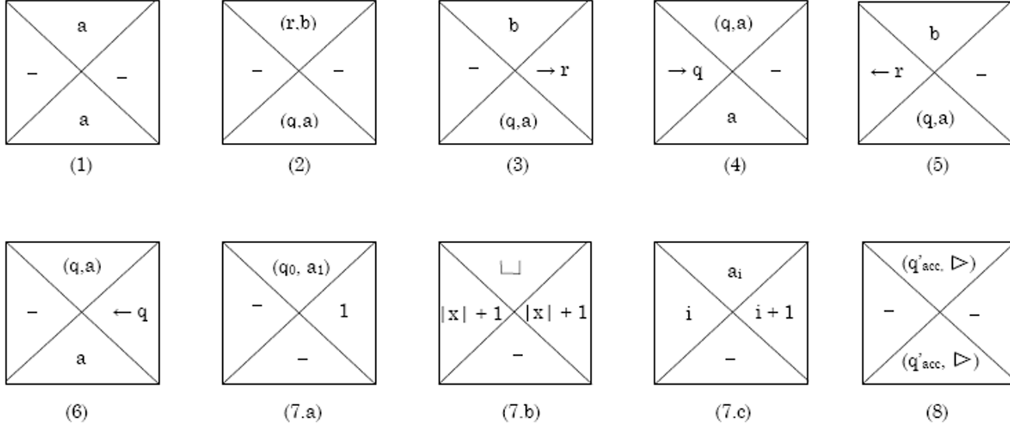

Figure 3.1: The tiles used in the proof of the EXPSPACE-complenetess of the 2^n-corridor tiling problem.

right, the current symbol is adjusted and the right side of the tile is marked with the new state and a right arrow to denote the right movement (item 3). Furthermore for every symbol, except $\triangleright$, it is possible that the tape head will arrive from the left, with a right movement, at that symbol (item 4). We don't create these tiles for the symbol $\triangleright$ since the definition of our Turing machine does not allow the $\triangleright$ to occur at any other position but the first. The tiles in items 5 and 6 do exactly the same for a left movement of the tape head. Here, we don't create a tile for the symbol $\triangleleft$, since $\triangleleft$ it is impossible for M' to reach the $\triangleleft$ at the $2^{(|x|)^k} + 1$th position of the tape. The tiles in item 7 take care of the input. Its first tile, $< _, (q_0, a_1), 1, _ >$, is o_{bot} and must be located at the first position of the first row. The only tile that fits at the left side is $< 1, a_2, 2, _ >$ and so on until the tile $< |x|, a_{|x|}, |x| + 1, _ >$. After this, only the tile $< |x|+1, \sqcup, |x|+1, _ >$ can complete this row. The top sides of this first row are $(q_0, a_1)a_2 \cdots a_n \sqcup\sqcup \cdots$, exactly the starting configuration of M'. Finally, when the tape head arrives at q'_{accept} at the first position of the tape, the tile in item 8 will copy $(q'_{accept}, \triangleright)$. Since o_{top} is equal to this tile, the tiling will then be accepted.

Finally, we say that $n = 4 + 14|x|^k$. This number will be explained later. We have to show that there exists a 2^n-corridor tiling if and only if M' (and thus M) accepts x. Suppose that there exists a 2^n-corridor tiling. By the construction of the tiles and because o_{bot} has to be the tile at the first position of the first row, we know that the top sides of the bottom row encode the start configuration of M'. The next row is placed on top of this row and must, by construction of the tiles, contain a valid configuration that can follow the first configuration. This holds for all rows of the tiling and thus

will the m rows of the tiling encode a valid series of configurations $c_1 \cdots c_m$. Since the first tile of the top row must be o_{top}, we have constructed a tiling that encodes a valid accepting run of M' on x, and thus x is also accepted by M.

There is however one problem with this explanation. It is possible that a tile of type 4 or 6 is be placed at the leftmost or rightmost position of the tape, hereby inserting a new (phantom) head. It is obvious that if this happens, we can not be sure anymore that a correct tiling encodes a valid run of M'. This is solved by the special $\rhd$ and $\lhd$ tokens. By the definition of our TM, the $\rhd$-token is always located at the first position of the tape. Since in item 6 no tile of the form $<\rightarrow q, (q, \rhd), _, \rhd>$ is constructed, it is impossible that a phantom head will appear at the left side of the tape.

At the right side, the situation is a bit more complicated. It is possible that a phantom head will appear there. However, since in item 4 no tiles of the form $< _, (q, \lhd), \leftarrow q, \lhd>$ are constructed, the $\lhd$-token will block every possible phantom head. The $\lhd$-token is placed at the $2^{(|x|)^k} + 1$th position, so there is enough place at the left of $\lhd$ to simulate M'. One little problem still arises, it takes a number of steps of M' to place the $\lhd$ at the $2^{(|x|)^k} + 1$th position. Until it is placed, we must be sure that a phantom head cannot have reached one of the first $2^{(|x|)^k} + 1$ positions. The following proposition gives us an upper bound on the number of steps it takes to put the $\lhd$ into place.

Proposition 3.9. *The TM M', places $\lhd$ on the $2^{(|x|)^k} + 1$th position in less than $2^{2+12|x|^k}$ steps.*

Proof. We investigate the steps described in proposition 3.8, that place the $\lhd$-token:

- Write the number and $\lhd$ and move the tape head back to the last bit of the number: $|x|^k + 2$ steps.
- Move the $\lhd$ to its $2^{(|x|)^k} + 1$th position: To check if the number is equal to $|x| + (|x|)^k + 2$ and diminish it by one, we never need more than $4|x|^k$ steps. The number of steps to reach and move $\lhd$ becomes bigger every time. We know that we never have to repeat these two operations more than $2^{(|x|)^k}$ times. This gives us $2^{(|x|)^k} 4|x|^k + 2(2 + 3 + \ldots + (2^{(|x|)^k} + 1))$ steps.

If we sum these, and use (very rough) substitutions, we can easily see that this sum is always smaller than $2^{2+12|x|^k}$. $\square$

Using proposition 3.9, we have defined n as $4 + 14|x|^k$, which forces the rows of the tiling to have a width of $2^{4+14|x|^k}$. This solves our problem of the phantom heads because a phantom head can only be born at the rightmost position of a row. Afterwards, it can only move one place to the left for each new row. So, by the time the first phantom head can reach the $2^{(|x|)^k} + 1$th tile, the $\lhd$ is already put in place, and blocks the phantom head.

We can now really show that if there exists a valid 2^n-corridor tiling, M' (and thus M) accepts x. The only difference with our first argument, is that not the whole rows of our tiling encode configurations of M', but only the $2^{(|x|)^k} + 1$ first tiles of every row encode a configuration of M'. By our definition of n and the use of the symbols $\rhd$ and $\lhd$, we have seen that we can guarantee that no phantom head can occur in the first $2^{(|x|)^k} + 1$ positions of our tiling. Therefore, our original reasoning is correct again and a valid tiling encodes an accepting run of M' on x.

Conversely, suppose that x is accepted by M. We know, by construction, that x is also accepted by M'. We can now easily simulate the accepting run of M' on x by a tiling. The first row of the tiling consists of the tiles that are specially made to encode the start configuration. For all the other rows, the first $(2^{(|x|)^k} + 1)$ tiles represent the successive configurations. The tiles that come after the first $(2^{(|x|)^k} + 1)$ tiles can all be filled with $< _, \sqcup, _, \sqcup >$ tiles. Finally, the accepting configuration on the top row is copied using the tile $< _, (q'_{accept}, \rhd), _, (q'_{accept}, \rhd) >$ as the first tile and the tiles $< _, a, _, a >$ for every other position on the tape with the symbol $a \in \Sigma$ written on it.

This is a valid tiling. Furthermore is the first tile of the bottom row equal to $o_{bot} =< _, (q_0, a_1), 1, _ >$, and is the first tile of the top row equal to $o_{top} =< _, (q'_{accept}, \rhd), _, (q'_{accept}, \rhd) >$. This gives us a valid 2^n-corridor tiling.

The only thing left to show is that we can do this reduction in logarithmic space. We have already shown that the adaptations to M can be done in logarithmic space. We have to show that the input items for the 2^n-corridor tiling problem can be constructed in logarithmic space.

The integer n is a polynomial that can be constructed by successive additions and multiplications of other polynomial factors. These numbers can all be stored in logarithmic space. The addition and multiplication of these polynomials also does not require more than logarithmic space.

Finally, we investigate the different tile-types. For the tiles constructed in items 1 to 6, we only have to keep track of one pointer to the set of states and/or one pointer to the set of symbols. These pointers only require logarithmic space. Sometimes, we also have to check the transition function. This can also be done by running through the transition function and thus using logarithmic space. For the tiles in item 7 we have to keep the variable i. This variable can never be bigger than $|x|$ and can thus be stored using

$\lceil \log |x| \rceil$ space. The tile in item 8 and the assignment of o_{bot} and o_{top} require constant space.

□

Chapter 4

Regular expressions

4.1 Regular expressions

Before defining regular expressions, we describe some basic terms concerning string languages. The set of all strings over an alphabet Σ 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 $\{x \cdot x' \mid x \in L, x' \in L'\}$. We abbreviate $L \cdot L \cdots L$ (i times) by L^i, and define $L^0 = \{\varepsilon\}$. The shuffle operator & is defined inductively as:

- $w \,\&\, \varepsilon = \varepsilon \,\&\, w = \{w\}$, for $w \in \Sigma^*$; and
- $ax \,\&\, bw = a(x \,\&\, bw) \cup b(ax \,\&\, w)$, for $x, w \in \Sigma^*$ and $a, b \in \Sigma$.

Definition 4.1. *The set of* regular expressions *over Σ, denoted by RE, is defined as follows:*

- *$\emptyset$, ε, and every Σ-symbol is a regular expressions; and*
- *when r and s are regular expressions, $r \cdot s$, $r + s$, r^*, $r^{k..l}(k, l \in \mathbb{N}$ and $k \leq l)$, and $r \,\&\, s$ are also regular expressions.*

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

- *$L(\emptyset) = \emptyset$;*
- *$L(\varepsilon) = \{\varepsilon\}$;*
- *$L(a) = \{a\}$;*
- *$L(r \cdot s) = L(r) \cdot L(s)$;*

- $L(r + s) = L(r) \cup L(s)$;
- $L(r^*) = \bigcup_{i=0}^{\infty} L(r)^i$;
- $L(r^{k..l}) = \bigcup_{i=k}^{l} L(r)^i$; *and*
- $L(r \,\&\, s) = L(r) \,\&\, L(s) = \bigcup_{x_1 \in L(r), x_2 \in L(s)} x_1 \,\&\, x_2$.

The $\cdot, +, *$-operators are the standard operators for regular expressions. The expressions of the form $r^{k..l}$ are called numerical occurrence indicators. Here, the expression r must occur at least k and at most l times. We will call k the *minimum bound* and l the *maximum bound*. The last operator, &, is the shuffle operator. It allows the words of its operands to be shuffled.

Example 4.2. *The regular expression $a^{0..5} \,\&\, bc$ describes strings with exactly one b, exactly one c, and zero to five a's. These characters can occur in any order, the only restriction is that the b comes before the c. Examples of strings defined by $a^{0..5} \,\&\, bc$ are $abaacaa$, $baca$, and bc. The string $acab$ is not defined by $a^{0..5} \,\&\, bc$.* □

The size of a numerical occurrence indicator with lower and upper bounds k and l is defined as $\lceil \log k \rceil + \lceil \log l \rceil$, the space required to store k and l in binary. The *size* of a regular expression r over Σ is the number of symbols, Σ-symbols and operator symbols, occurring in r plus the sum of the sizes of the numerical occurrence indicators in r. By $r?$ and r^+, we abbreviate the expressions $r + \varepsilon$ and $r \cdot r^*$, respectively. Often, we omit the concatenation symbol $\cdot$ and replace $r \cdot s$ by rs and sometimes we denote $x \in L(r)$ simply by $x \in r$.

We will often consider subsets of the here defined regular expressions. By RE(S), S being a list of operators, we denote all regular expressions that can be formed using only the operators in S. The numerical occurrence indicator is represented by #. For example, RE($\cdot, +, *$) denotes the "standard" set of regular expressions and RE($\cdot, +, *, \#, \&$) represents the set of all regular expressions.

As mentioned in the introduction, we will also study classes of simpler regular expressions. These CHAREs are defined as follows:

Definition 4.3. *A* base symbol *is a regular expression $s, s^*, s^+, s?$, or $s^{k..l}$ ($k, l \in \mathbb{N}, k \leq l$), where s is a non-empty string; a* factor *is of the form $e, e^*, e^+, e?$ or $e^{k..l}$ ($k, l \in \mathbb{N}, k \leq l$) where e is a disjunction of base symbols of the same kind. That is, e is of the form $(s_1 + \cdots + s_n), (s_1^* + \cdots + s_n^*), (s_1^+ + \cdots + s_n^+), (s_1? + \cdots + s_n?)$, or $(s_1^{k_1..l_1} + \cdots + s_n^{k_n..l_n})$, where $n \geq 0$ and $s_1, \ldots, s_n$ are non-empty strings. A* chain regular expression *(CHARE) is $\emptyset, \varepsilon$, or a sequence of factors.*

The regular expressions $((abc)^* + b^*)(a + b)?(ab)^{1..2}(ac + b)^*(a^{0..2} + c^{2..4})^*$ is a chain regular expression. The expression $(a+b)+(a^*b^*)$, however, is not.

We use a uniform syntax to denote subclasses of chain regular expressions by specifying the allowed factors. We distinguish whether the string s of a base symbol consists of a single symbol (denoted by a) or a string (denoted by w) and whether it is extended by $*, +, ?$, or $\#$. Furthermore, we distinguish between factors with one disjunct or with arbitrarily many disjuncts: the latter is denoted by $(+\cdots)$. Finally, factors can again be extended by $*, +, ?$, or $\#$. A list of possible factors and their corresponding expressions are listed in Table 4.1.

We denote subclasses of chain regular expressions by CHARE(X), where X is a list of the allowed factors. For example, we write CHARE($(+a)^*, w?$) for the set of regular expressions $e_1 \cdots e_n$ where every e_i is either (i) $(a_1 + \cdots + a_m)^*$ for some $a_1, \ldots, a_m \in \Sigma$ and $m \geq 1$, or (ii) $w?$ for some $w \in \Sigma^+$. If $A = \{a_1, \ldots, a_n\}$ is a set of symbols, we often denote $(a_1 + \ldots + a_n)$ simply by A. We denote the class of all chain regular expressions by CHARE(S). Finally, we denote the class of all chain regular expressions without numerical occurrence indicators as CHARE($S \backslash \{\#\}$). This is the class of chain regular expressions as defined by Martens et al. [MNS04].

4.2 Automata for regular expressions

We need a way to test if a string x is accepted by a regular expression r. We use automata to do this.

Definition 4.4. *A* deterministic finite automaton (DFA) *is a tuple* $A = (Q, \Sigma, q_0, q_f, \delta)$ *where*

- Q *is the finite set of states;*
- Σ *is the finite alphabet;*
- $\delta : Q \times \Sigma \to Q$ *is the transition function;*
- $q_0 \in Q$ *is the start state; and*
- $q_f \in Q$ *is the final state.*

A *run* of a DFA A on a string $x = a_1 \cdots a_n$ is a function $\lambda : \{1, \ldots, n + 1\} \to Q$ such that

- $\lambda(1) = q_0$; and

factor	**expression**
a	a
a^*	a^*
a^+	a^+
$a?$	$a?$
$a^{k..l}$	$a^{\#}$
w^*	w^*
w^+	w^+
$w?$	$w?$
$w^{k..l}$	$w^{\#}$
$(a_1 + \cdots + a_n)$	$(+a)$
$(a_1 + \cdots + a_n)^*$	$(+a)^*$
$(a_1 + \cdots + a_n)^+$	$(+a)^+$
$(a_1 + \cdots + a_n)?$	$(+a)?$
$(a_1 + \cdots + a_n)^{k..l}$	$(+a)^{\#}$
$(a_1^* + \cdots + a_n^*)$	$(+a^*)$
$(a_1^+ + \cdots + a_n^+)$	$(+a^+)$
$(a_1? + \cdots + a_n?)$	$(+a?)$
$(a_1^{k_1..l_1} + \cdots + a_n^{k_n..l_n})$	$(+a^{\#})$
$(w_1 + \cdots + w_n)$	$(+w)$
$(w_1 + \cdots + w_n)^*$	$(+w)^*$
$(w_1 + \cdots + w_n)^+$	$(+w)^+$
$(w_1 + \cdots + w_n)?$	$(+w)?$
$(w_1 + \cdots + w_n)^{k..l}$	$(+w)^{\#}$
$(w_1^* + \cdots + w_n^*)$	$(+w^*)$
$(w_1^+ + \cdots + w_n^+)$	$(+w^+)$
$(w_1? + \cdots + w_n?)$	$(+w?)$
$(w_1^{k_1..l_1} + \cdots + w_n^{k_n..l_n})$	$(+w^{\#})$

Table 4.1: Possible factors in chain regular expressions and how they are denoted. ($a, a_i \in \Sigma$, $w, w_i \in \Sigma^*$, $k, l, k_i, l_i \in \mathbb{N}$)

- for all $i \in \{1, \cdots, n\}$, $\lambda(i+1) = \delta(\lambda(i), a_i)$.

A run is *accepting* if $\lambda(n+1) = q_f$. A string is accepted if there is an accepting run for it. The set of all strings accepted by A is denoted by $L(A)$.

We can add non-determinism in the conventional way. Denote $\Sigma \cup \{\varepsilon\}$ by Σ_ε.

Definition 4.5. *A* non-deterministic finite automaton (NFA) *is a deterministic finite automaton, where the transition function δ is of the form $\delta : Q \times \Sigma_\varepsilon \to P(Q)$, $P(Q)$ being the powerset of Q.*

On input a string x, suppose that we can write x as $x = a_1 \cdots a_n$, $a_i \in \Sigma_\varepsilon$. A *run* of a NFA A on x is a function $\lambda : \{1, \cdots, n+1\} \to Q$ such that

- $\lambda(1) = q_0$; and
- for all $i \in \{1, \cdots, n\}$, $\lambda(i+1) \in \delta(\lambda(i), a_i)$.

A run is *accepting* if $\lambda(n+1) = q_f$. A string is accepted if there is an accepting run for it. The set of all strings accepted by A is denoted by $L(A)$.

Note that in the definition of our automata, we allow only one start and only one accept state. In some other definitions of string automata, sets of start or final states are allowed. This, however, does not add power to the automata. We have chosen one start and final state because this makes some proofs easier to understand.

Theorem 4.6. *For any RE$(\cdot, +, *)$-expression r, we can construct a NFA A such that $L(r) = L(A)$, in time polynomial in the size of r.*

Proof. We use a simple construction to transform a regular RE$(\cdot, +, *)$ expression r into an equivalent NFA $A = (Q, \Sigma, q_0, q_f, \delta)$. This construction is a slight adaptation of the construction used by Sipser ([Sip96]).

We construct the automaton recursively as follows:

- If $r = \emptyset$: $Q = \{q_r, f_r\}, q_0 = q_r, q_f = f_r$.
- If $r = \varepsilon$: $Q = \{q_r, f_r\}, q_0 = q_r, q_f = f_r, \delta(q_r, \varepsilon) = f_r$.
- If $r = a, a \in \Sigma$: $Q = \{q_r, f_r\}, q_0 = q_r, q_f = f_r, \delta(q_r, a) = f_r$.
- If $r = e_1 \cdot e_2$: say $A(e_i) = (Q_i, \Sigma, \delta_i, q_i, f_i), i = 1, 2$: $Q = Q_1 \cup Q_2, q_0 = q_1, q_f = f_2, \delta = \delta_1 \cup \delta_2 \cup \delta(f_1, \varepsilon) = q_2$.
- If $r = e_1 + e_2$: say $A(e_i) = (Q_i, \Sigma, \delta_i, q_i, f_i), i = 1, 2$: $Q = Q_1 \cup Q_2 \cup \{q_r, f_r\}, q_0 = q_r, q_f = f_r, \delta = \delta_1 \cup \delta_2 \cup \delta(q_r, \varepsilon) = q_1 \cup \delta(q_r, \varepsilon) = q_2 \cup \delta(f_1, \varepsilon) = f_r \cup \delta(f_2, \varepsilon) = f_r$.

- If $r = e^*$: say $A(e) = (Q_e, \Sigma, \delta_e, q_e, f_e)$: $Q = Q_1 \cup \{q_r, f_r\}, q_0 = q_r, q_f = f_r, \delta = \delta_e \cup \delta(q_r, \varepsilon) = q_e \cup \delta(q_r, \varepsilon) = f_r \cup \delta(f_e, \varepsilon) = q_e \cup \delta(f_e, \varepsilon) = f_r$.

This construction adds a constant number of new states and entries in the transition function for every symbol in r. Since there are a polynomial number of symbols in r, this construction operates in polynomial time.

□

The recursive construction of the NFA is illustrated in Figure 4.1.

We will now expand the capabilities of the NFA with counters, and allow the automaton to be in more than one state at once. This gives us the possibility to translate a RE$(\cdot, *, +, \#, \&)$ directly into such an extended non-deterministic finite automaton (ENFA). Kilpelainen already mentioned the idea of extending NFAs with counters ([KT03]). Jedrzejowicz and Szepietowski fully defined shuffle automata ([JS01]). Here, we expand these shuffle automata with a counter mechanism.

In an ENFA, every state has a *mincount* and a *maxcount* and maintains a *counter*. Every time a state is visited, its counter is augmented by 1. A state also has two different types of out-edges, *in* and *reset* edges. For a state q, we are allowed to follow an *in* edge if $counter(q) < maxcount(q)$ and are allowed to follow a reset edge if $counter(q) \geq mincount(q)$ and $counter(q) \leq maxcount(q)$. If we follow a reset edge, we also reset the counter of that state to zero. These counters help us to immediately translate the numerical occurrence indicator into an ENFA.

Furthermore, we add transitions that go from one (split) state to two new states. Conversely, there are also transitions that go from two states to one (merge) state. Therefore, a configuration of an ENFA will be a set of states, instead of just one state.

An ENFA is defined as follows:

Definition 4.7. *An* extended non-deterministic finite automaton (ENFA) *is a five-tuple* $(Q, \Sigma, q_0, q_f, \delta)$ *in which:*

- $Q = Q_N \uplus Q_S \uplus Q_M$ *is the finite set of states, and with every state* $q \in Q$ *integers* $mincount(q)$ *and* $maxcount(q)$*, where* $mincount(q) \leq maxcount(q)$*, are associated. Here,* Q_N *is the finite set of normal states,* Q_S *is the finite set of split states, and* Q_M *is the finite set of merge states;*

- Σ *is a finite alphabet;*

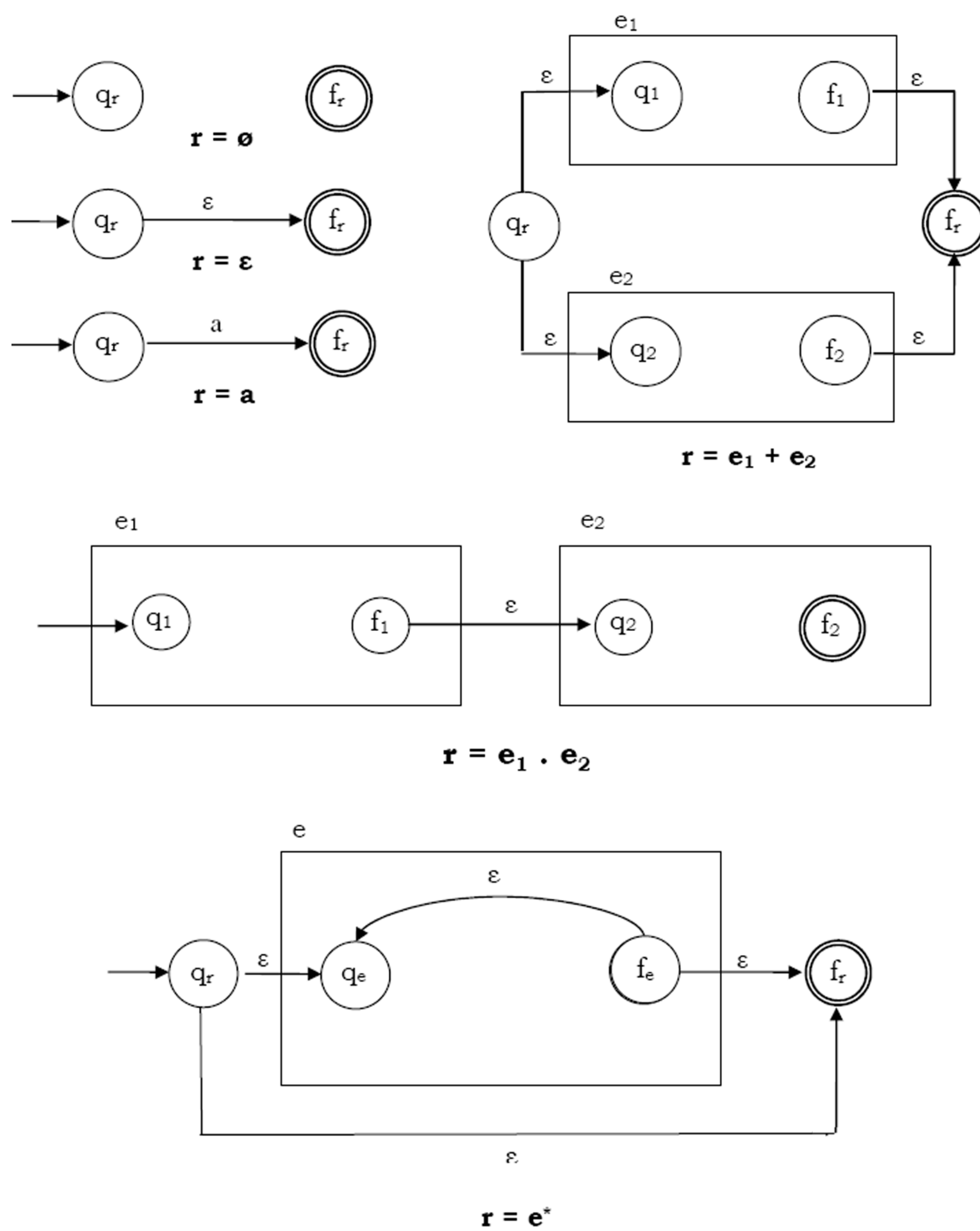

Figure 4.1: The construction of an NFA which is equivalent to a given regular expression.

- $q_0 \in Q$ *is the start state;*
- $q_f \in Q$ *is the accept state; and*
- $\delta : \Sigma_\varepsilon \times ((Q \times \{IN, RESET\} \times Q) \cup (Q_S \times \{RESET\} \times Q \times Q) \cup (Q \times Q \times \{RESET\} \times Q_M)$ *is a finite set of transitions.*

Furthermore, during the computation, we associate an integer $count(q)$ *with every state* q*.*

A *partial configuration* ϕ of an ENFA automaton $A = (Q, \Sigma, q_0, q_f, \delta)$ is a $(|Q|+1)$-tuple which contains a state q, denoted by $state(\phi)$, and an integer for every $p \in Q$, denoted by $count(\phi, p)$. If $Q = \{q_1, \ldots, q_n\}$, we represent the partial configuration ϕ as $(state(\phi), q_1 = count(\phi, q_1), \ldots, q_n = count(\phi, q_n))$. A *configuration* γ of A is a finite set of partial configurations.

A configuration γ_2 *immediately follows* a configuration γ_1 using a transition $z \in \delta$, denoted by $\gamma_1 \Rightarrow^z \gamma_2$, if the following conditions are satisfied:

- If $z = (a, p, IN, q)$, $a \in \Sigma_\varepsilon$, $p, q \in Q$, then one partial configuration goes from state p to state q, the count of p in that partial configuration remains the same, and the count of q in that partial configuration is incremented.

  Formally, there must exist partial configurations $\phi \in \gamma_1$, $\varphi \in \gamma_2$, such that $\gamma_2 = (\gamma_1 \backslash \{\phi\}) \cup \{\varphi\}$, and $state(\phi) = p$, $state(\varphi) = q$, $count(\phi, p) < maxcount(p)$, $count(\varphi, q) = count(\phi, q) + 1$, and $count(\phi, q') = count(\varphi, q')$, for all $q' \in Q, q' \neq q$.

- If $z = (a, p, RESET, q)$, $a \in \Sigma_\varepsilon$, $p, q \in Q$, then one partial configuration goes from state p to state q, the count of p in that partial configuration is reset, and the count of q in that partial configuration is incremented.

  Formally, there must exist partial configurations $\phi \in \gamma_1$, $\varphi \in \gamma_2$, such that $\gamma_2 = (\gamma_1 \backslash \{\phi\}) \cup \{\varphi\}$, and $state(\phi) = p$, $state(\varphi) = q$, and $count(\phi, p) \geq mincount(p)$, $count(\phi, p) \leq maxcount(p)$, $count(\varphi, q) = count(\phi, q) + 1$, $count(\varphi, p) = 0$, and $count(\phi, q') = count(\varphi, q')$, for all $q' \in Q, q' \neq q, q' \neq p$.

- If $z = (a, p, RESET, q_1, q_2)$, $a \in \Sigma_\varepsilon$, $p \in Q_S$, $q_1, q_2 \in Q$, then the partial configuration which is in state p is split in two partial configurations, which are in states q_1 and q_2. The count of p in both partial configurations is reset, and the count of q_1 (resp. q_2) is incremented in the partial configuration of state q_1 (resp. q_2).

Formally, there must exist partial configurations $\phi \in \gamma_1$, $\varphi_1, \varphi_2 \in \gamma_2$, such that $\gamma_2 = (\gamma_1 \backslash \{\phi\}) \cup \{\varphi_1, \varphi_2\}$, and $state(\phi) = p$, $state(\varphi_1) = q_1$, $state(\varphi_2) = q_2$, and $count(\phi, p) \geq mincount(p)$, $count(\phi, p) \leq maxcount(p)$, $count(\varphi_1, q_1) = count(\phi, q_1){+}1$, $count(\varphi_2, q_2) = count(\phi, q_2){+}1$, $count(\varphi_1, p) = count(\varphi_2, p) = 0$, and $count(\phi, q') = count(\varphi_1, q')$, for all $q' \in Q, q' \neq q_1, q' \neq p$, and $count(\phi, q') = count(\varphi_2, q')$, for all $q' \in Q, q' \neq q_2, q' \neq p$.

- If $z = (a, p_1, p_2, RESET, q)$, $a \in \Sigma_\varepsilon$, $p_1, p_2 \in Q$, $q \in Q_M$, then two partial configurations which are in states p_1 and p_2 are merged into one partial configuration which is in state q. The counters of p_1 and p_2 are reset, and the counter of q is incremented.

  Formally, there must exist partial configurations $\phi_1, \phi_2 \in \gamma_1$, $\varphi_1 \in \gamma_2$, such that $\gamma_2 = (\gamma_1 \backslash \{\phi_1, \phi_2\}) \cup \{\varphi\}$, and $state(\phi_1) = p_1$, $state(\phi_2) = p_2$, $state(\varphi) = q$, and $count(\phi, p_1) \geq mincount(p_1)$, $count(\phi, p_1) \leq maxcount(p_1)$, $count(\phi, p_2) \geq mincount(p_2)$, $count(\phi, p_2) \leq maxcount(p_2)$, and for every $p' \in Q$, $p' \neq p_1$, $p' \neq p_2$, $count(\phi_1, p') = count(\phi_2, p')$, and $count(\varphi_1, q) = count(\phi_1, q) + 1$, $count(\varphi_1, p_1) = 0$, $count(\varphi_1, p_2) = 0$, and $count(\phi, q') = count(\varphi_1, q')$, for all $q' \in Q, q' \neq p_1, q' \neq p_2, q' \neq q$.

We now extend this notion to a sequence of transitions $\theta = z_1 \cdots z_n$. We say that a configuration γ_n *follows* a configuration γ_0 using a sequence of transitions $\theta \in \delta^*$, denoted by $\gamma_0 \Rightarrow^\theta \gamma_n$, if there exist configurations $\gamma_0, \ldots, \gamma_n$ such that $\gamma_{i-1} \Rightarrow^{z_i} \gamma_i$, for every $0 < i \leq n$.

For $\theta = z_1 \cdots z_n$ a sequence of transitions, we denote by $label(\theta)$ the string associated to θ. That is, $label : \delta \to \Sigma$ is defined by $label(z) = a$, for $z = (a, p, IN, q)$, $z = (a, p, RESET, q)$, $z = (a, p, RESET, q, r)$, or, $z = (a, p, q, RESET, r)$ where $p, q, r \in Q$, and $a \in \Sigma_\varepsilon$. Then, for two sequences of transitions θ_1, θ_2, we recursively define the label of $\theta_1 \cdot \theta_2$ as $label(\theta_1 \cdot \theta_2) = label(\theta_1) \cdot label(\theta_2)$.

Example 4.8. *If θ is equal to $(c, q_1, IN, q_2)(\varepsilon, q_2, IN, q_3)(d, q_3, RESET, q_1)$, then $label(\theta) = cd$. Note that the labeling function* ignores *ε-transitions, which allows us to follow at any time a ε-transition in the computation.* □

A configuration γ_0 is the initial configuration of A if it is a singleton $\{\phi\}$, where $state(\phi) = q_0$, $count(\phi, q_0) = 1$, and $count(\phi, q) = 0$ for all states $q \neq q_0$. That is, the computation starts with the start state q_0 and all counters, but the counter of q_0, set to zero. The counter of the start state is set to one, because the count of the current state can never be equal to zero. Indeed, by beginning the computation, we have in fact entered the

start state, which has set its counter set to one. A configuration γ_n is an accepting configuration if it is a singleton $\{\varphi\}$, where $state(\varphi) = q_f$, and $count(\varphi, q_f) \geq mincount(q_f)$. We only accept if the current state is the accept state, and the count of the accept state is greater than its minimum count. Note that we don't require the count of the accept state to be less than or equal the maxcount. This is immediate since the count of a state can never be bigger than its maxcount. This follows from the fact that when the count is equal to its maxcount, we are obligated to follow a RESET edge, whereby we reset the count of the state to zero.

A string x is *accepted* by A if there exist configurations γ_0 and γ_n, and a sequence of transitions $\theta \in \delta^*$, such that

- γ_0 is the initial configuration of A;
- γ_n is an accepting configuration of A;
- $label(\theta) = x$; and
- $\gamma_0 \Rightarrow^\theta \gamma_n$.

The sequence of transitions θ is also referred to as an *accepting run of A on x*. As usual, we denote by $L(A)$ the set of strings accepted by A.

Given a RE$(\cdot, *, +, \#, \&)$-expression r, we can construct an equivalent ENFA $A_r = (Q, \Sigma, q_0, q_f, \delta)$ recursively as follows, where for every state q, $mincount(q) = maxcount(q) = 1$ unless stated otherwise.

- If $r = \emptyset$, then $Q = \{q_r, f_r\}, q_r, f_r \in Q_N, q_0 = q_r, q_f = f_r$.
- If $r = \varepsilon$, then $Q = \{q_r, f_r\}, q_r, f_r \in Q_N, q_0 = q_r, q_f = f_r$, $\delta = \{(\varepsilon, q_r, RESET, f_r)\}$.
- If $r = a, a \in \Sigma$, then $Q = \{q_r, f_r\}, q_r, f_r \in Q_N, q_0 = q_r, q_f = f_r, \delta = \{(a, q_r, RESET, f_r)\}$.
- If $r = e_1 + e_2$, then say $A(e_i) = (Q_i, \Sigma, q_i, f_i, \delta_i), i = 1, 2$: $Q = Q_1 \cup Q_2 \cup \{q_r, f_r\}, q_r, f_r \in Q_N, q_0 = q_r, q_f = f_r, \delta = \delta_1 \cup \delta_2 \cup \{(\varepsilon, q_r, RESET, q_1), (\varepsilon, q_r, RESET, q_2), (\varepsilon, f_1, RESET, f_r), (\varepsilon, f_2, RESET, f_r)\}$.
- If $r = e_1 \cdot e_2$, then say $A(e_i) = (Q_i, \Sigma, q_i, f_i, \delta_i), i = 1, 2$: $Q = Q_1 \cup Q_2, q_0 = q_1, q_f = f_2, \delta = \delta_1 \cup \delta_2 \cup \{(\varepsilon, f_1, RESET, q_2)\}$.
- If $r = e^*$, then say $A(e) = (Q_e, \Sigma, q_e, f_e, \delta_e)$: $Q = Q_e \cup \{q_r, f_r\}, q_r, f_r \in Q_N, q_0 = q_r, q_f = f_r, \delta = \delta_e \cup \{(\varepsilon, q_r, RESET, q_e), (\varepsilon, f_e, RESET, f_r), (\varepsilon, f_e, RESET, q_e), (\varepsilon, q_r, RESET, f_r)\}$.

- If $r = e^{k..l}$, then say $A(e) = (Q_e, \Sigma, q_e, f_e, \delta_e)$: $Q = Q_e \cup \{q_r, f_{r1}, f_{r2}\}, q_r, f_{r1}, f_{r2} \in Q_N, q_0 = q_r, q_f = f_{r2}, mincount(f_{r_2}) = k, maxcount(f_{r_2}) = l, \delta = \delta_e \cup \{(\varepsilon, q_r, RESET, q_e), (\varepsilon, f_e, RESET, f_{r1}), (\varepsilon, f_{r1}, IN, q_e), (\varepsilon, f_{r1}, RESET, f_{r2})\}$. If $k = 0$, we add $(\varepsilon, q_r, RESET, f_{r2})$ to this union.

- If $r = e_1 \,\&\, e_2$, then say $A(e_i) = (Q_i, \Sigma, q_i, f_i, \delta_i), i = 1, 2$: $Q = Q_1 \cup Q_2 \cup \{q_r, f_r\}, q_r \in Q_S, f_r \in Q_M, q_0 = q_r, q_f = f_r, \delta = \delta_1 \cup \delta_2 \cup \{(\varepsilon, q_r, RESET, q_1, q_2), (\varepsilon, f_1, f_2, RESET, f_r)\}$.

For all operators but the numerical occurrence and shuffle operator, this construction is the same as that for NFAs (Figure 4.1). The construction for the numerical occurrence and shuffle operator is illustrated in Figure 4.2. Here, every edge that does not have an IN or RESET marking, is a RESET edge, and every state that does not have a mincount or maxcount, has mincount = maxcount = 1.

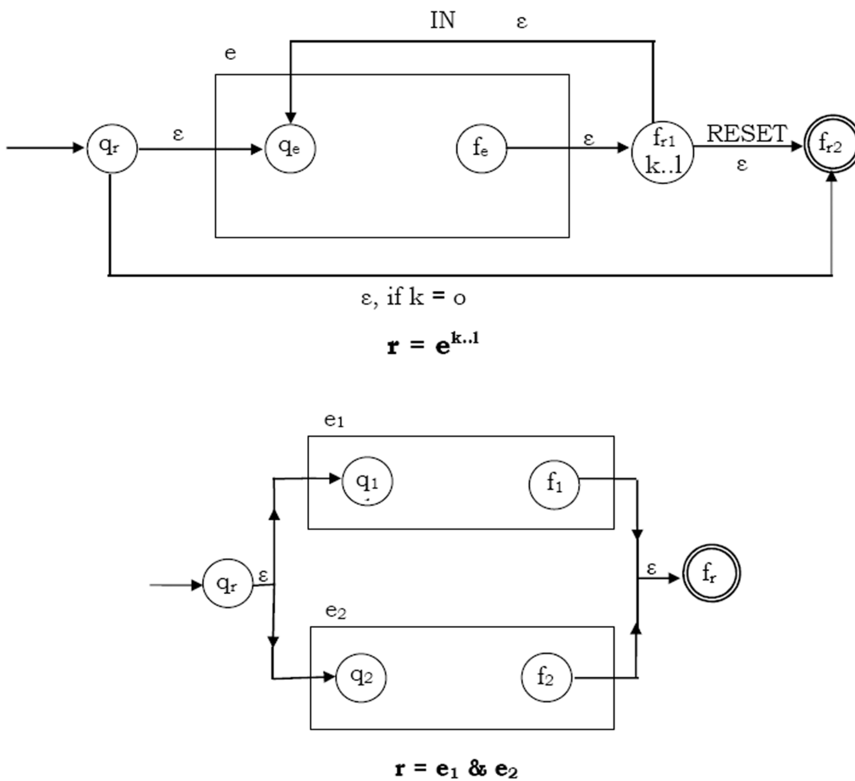

Figure 4.2: The construction used for the numerical occurence and shuffle operator.

Example 4.9. *The regular expression* $(a+b)^{1..3}\,\&\,c^*$ *is translated in the following ENFA* $A_r = (\{q_1,\ldots,q_{15}\},\{a,b,c\},q_1,q_{15},\delta)$, *where*

$$\delta = \{ \begin{array}{lll} (\varepsilon,q_1,RESET,q_2,q_{11}), & (\varepsilon,q_2,RESET,q_3), & (\varepsilon,q_3,RESET,q_4), \\ (\varepsilon,q_3,RESET,q_6), & (a,q_4,RESET,q_5), & (b,q_6,RESET,q_7), \\ (\varepsilon,q_5,RESET,q_8), & (\varepsilon,q_7,RESET,q_8), & (\varepsilon,q_8,RESET,q_9), \\ (\varepsilon,q_9,IN,q_3), & (\varepsilon,q_9,RESET,q_{10}), & (\varepsilon,q_{11},RESET,q_{12}), \\ (\varepsilon,q_{11},RESET,q_{14}), & (c,q_{12},RESET,q_{13}), & (\varepsilon,q_{13},RESET,q_{12}), \\ (\varepsilon,q_{13},RESET,q_{14}), & (\varepsilon,q_{10},q_{14},RESET,q_{15}) & \} \end{array}$$

and $mincount(q_9) = 1$, $maxcount(q_9) = 3$, $mincount(q) = maxcount(q) = 1$, *for every* $q \in Q$, $q \neq q_9$, *and* $Q_S = \{q_1\}, Q_M = \{q_{15}\}$, *and* $Q_N = \{q_2,\ldots,q_{14}\}$. *This automaton is illustrated in Figure 4.3. Again, we have omitted the RESET tags, and mincount and maxcount values when these are obvious.*

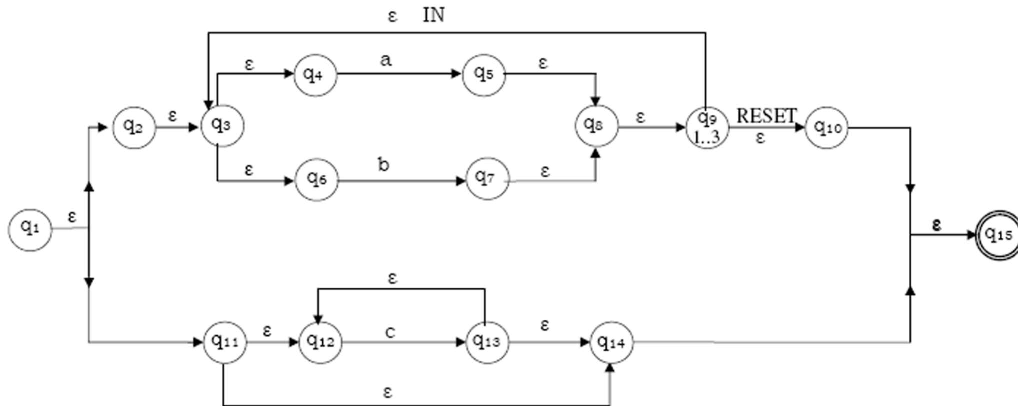

Figure 4.3: An ENFA for the regular expression $(a+b)^{1..3}\,\&\,c^*$.

An accepting run of A_r *on the string cabc is demonstrated in Table 4.9. Here, we only note the count of* q_9, *the only state for which the count is important. For example, the configuration* $\{(q_1, q_9 = 1), (q_2, q_9 = 2)\}$ *is in the states* q_1, *and* q_2, *and for* q_1 *the count of* q_9 *is one, for* q_2 *the count of* q_9 *is two.*

□

configuration	transition
$\{(q_1, q_9 = 0)\}$	$(\varepsilon, q_1, RESET, q_2, q_{11})$
$\{(q_2, q_9 = 0), (q_{11}, q_9 = 0)\}$	$(\varepsilon, q_{11}, RESET, q_{12})$
$\{(q_2, q_9 = 0), (q_{12}, q_9 = 0)\}$	$(c, q_{12}, RESET, q_{13})$
$\{(q_2, q_9 = 0), (q_{13}, q_9 = 0)\}$	$(\varepsilon, q_{13}, RESET, q_{12})$
$\{(q_2, q_9 = 0), (q_{12}, q_9 = 0)\}$	$(\varepsilon, q_2, RESET, q_3)$
$\{(q_3, q_9 = 0), (q_{12}, q_9 = 0)\}$	$(\varepsilon, q_3, RESET, q_4)$
$\{(q_4, q_9 = 0), (q_{12}, q_9 = 0)\}$	$(a, q_4, RESET, q_5)$
$\{(q_5, q_9 = 0), (q_{12}, q_9 = 0)\}$	$(\varepsilon, q_5, RESET, q_8)$
$\{(q_8, q_9 = 0), (q_{12}, q_9 = 0)\}$	$(\varepsilon, q_8, RESET, q_9)$
$\{(q_9, q_9 = 1), (q_{12}, q_9 = 0)\}$	$(\varepsilon, q_9, IN, q_3)$
$\{(q_3, q_9 = 1), (q_{12}, q_9 = 0)\}$	$(\varepsilon, q_3, RESET, q_6)$
$\{(q_6, q_9 = 1), (q_{12}, q_9 = 0)\}$	$(b, q_6, RESET, q_7)$
$\{(q_7, q_9 = 1), (q_{12}, q_9 = 0)\}$	$(\varepsilon, q_7, RESET, q_8)$
$\{(q_8, q_9 = 1), (q_{12}, q_9 = 0)\}$	$(\varepsilon, q_8, RESET, q_9)$
$\{(q_9, q_9 = 2), (q_{12}, q_9 = 0)\}$	$(\varepsilon, q_9, RESET, q_{10})$
$\{(q_{10}, q_9 = 0), (q_{12}, q_9 = 0)\}$	$(c, q_{12}, RESET, q_{13})$
$\{(q_{10}, q_9 = 0), (q_{13}, q_9 = 0)\}$	$(\varepsilon, q_{13}, RESET, q_{14})$
$\{(q_{10}, q_9 = 0), (q_{14}, q_9 = 0)\}$	$(\varepsilon, q_{10}, q_{14}, RESET, q_{15})$
$\{(q_{15}, q_9 = 0)\}$	

Table 4.2: An accepting run of A_r on *cabc*

4.3 Properties of extended NFAs

Theorem 4.10. *For any RE$(\cdot, +, *, \#, \&)$-expression r, there exists an ENFA A such that $L(r) = L(A)$.*

Proof. On input a RE$(\cdot, +, *, \#, \&)$-expression r, we construct the ENFA $A = (Q, \Sigma, q_0, q_f, \delta)$ as described above. We prove that $L(r) \subseteq L(A)$ (Lemma 4.11) and $L(A) \subseteq L(r)$ (Lemma 4.12). The theorem then follows. □

Lemma 4.11. *For any string x, if $x \in L(r)$, then $x \in L(A)$, and if $\gamma_n = \{\varphi\}$ is the accepting configuration of the accepting run of A on x, then $count(\varphi, q_f) = 1$ and $count(\varphi, q) = 0$, for every $q \in Q$, $q \neq q_f$.*

Proof. We prove this lemma inductively on the structure of r.

- If $r = \emptyset$, then $L(r) = \emptyset$, trivial.
- If $r = \varepsilon$, then $L(r) = \{\varepsilon\}$. The accepting run θ of A on ε is $\theta = (\varepsilon, q_r, RESET, f_r)$. The accepting configuration is $\gamma_n = \{\varphi\}$, where $\varphi = (q_f, q_r = 0, f_r = 1)$, which satisfies every condition.

- If $r = a$, $a \in \Sigma$, then $L(r) = \{a\}$. The accepting run θ of A on a is $\theta = (a, q_r, RESET, q_f)$. The accepting configuration is $\gamma_n = \{\varphi\}$, where $\varphi = (q_f, q_r = 0, q_f = 1)$, which satisfies every condition.

- If $r = e_1 + e_2$, suppose that $x \in L(e_1)$. Then, by induction, there exists a $\theta_x \in \delta^*_{e_1}$, and configurations $\gamma_{0,x} = \{\phi_x\}$ and $\gamma_{n,x} = \{\varphi_x\}$, such that $\gamma_{0,x} \Rightarrow^{\theta_x} \gamma_{n,x}$, $\gamma_{0,x}$ and $\gamma_{0,x}$ are initial and accepting configurations of A_{e_1} respectively, $count(\varphi_x, f_1) = 1$, and $count(\varphi_x, q) = 0$ for all other states q. Let $\theta = (\varepsilon, q_r, RESET, q_1) \cdot \theta_x \cdot (\varepsilon, f_1, RESET, q_f)$. Then there exist initial and accepting configurations $\gamma_0 = \{\phi\}$ and $\gamma_n = \{\varphi\}$ for A_r, such that $\gamma_0 \Rightarrow^{\theta} \gamma_n$, and $label(\theta_x) = label(\theta) = x$. Furthermore, since $\gamma_n = \{\varphi\}$ follows directly $\gamma_{n,x} = \{\varphi_x\}$ by transition $(\varepsilon, f_1, RESET, q_f)$, and $count(\varphi_x, f_1) = 1$ and $count(\varphi_x, q) = 0$ for all other states $q \in Q_1$, it follows that $count(\varphi, f_r) = 1$ and $count(\varphi, q) = 0$ for all other states $q \in Q$.

- If $r = e_1 \cdot e_2$ and $x \in L(r)$, then there exist $x_1 \in L(e_1)$ and $x_2 \in L(e_2)$ such that $x = x_1 \cdot x_2$. By induction, for $i = 1, 2$, there exists a $\theta_{x_i} \in \delta^*_{e_i}$, and configurations $\gamma_{0,x_i} = \{\phi_{x_i}\}$ and $\gamma_{n,x_i} = \{\varphi_{x_i}\}$, such that $label(\theta_{x_i}) = x_i$, $\gamma_{0,x_i} \Rightarrow^{\theta_{x_i}} \gamma_{n,x_i}$, γ_{0,x_i} and γ_{n,x_i} are initial and accepting configurations of A_{e_i} respectively, and $count(\varphi_{x_i}, f_i) = 1$ and $count(\varphi_{x_i}, q) = 0$ for all $q \in Q_i$, $q \neq f_i$. Let $\theta = \theta_{x_1} \cdot (\varepsilon, f_1, RESET, q_2) \cdot \theta_{x_2}$. Then there exist initial and accepting configurations $\gamma_0 = \{\phi\}$ and $\gamma_n = \{\varphi\}$ for A_r, such that $\gamma_0 \Rightarrow^{\theta} \gamma_n$, and $label(\theta) = label(\theta_{x_1}) \cdot label(\theta_{x_2}) = x$. Here, $\gamma_0 = \{\phi\}$ is the initial configuration of the accepting run of $A(e_1)$ on x_1, where $count(\phi, q) = 0$ is added for every $q \notin Q_1$, and $\gamma_n = \{\varphi\}$ is the accepting configuration of the accepting run of $A(e_2)$ on x_2, where $count(\varphi, q) = 0$ is added for every $q \notin Q_2$. By induction, the restrictions on the count values for the accepting configuration hold.

- If $r = e^*$, and $x \in L(r)$, then there are two possibilities. If $x = \varepsilon$, then there exist initial and accepting configurations for $A(r)$, such that for $\theta = (\varepsilon, q_r, RESET, f_r)$, $\gamma_0 \Rightarrow^{\theta} \gamma_n$, $label(\theta) = \varepsilon$, and the count conditions on φ hold. If $x \neq \varepsilon$, then there exist $x_i \in L(e)$, $i = 1, \ldots, n$ such that $x = x_1 \cdots x_n$. By induction, for every $i = 1, \ldots, n$, there exists a $\theta_{x_i} \in \delta^*_e$, and configurations $\gamma_{0,x_i} = \{\phi_{x_i}\}$ and $\gamma_{n,x_i} = \{\varphi_{x_i}\}$, such that $\gamma_{0,x_i} \Rightarrow^{\theta_{x_i}} \gamma_{n,x_i}$, γ_{0,x_i} and γ_{n,x_i} are initial and accepting configurations of A_e respectively, and $count(\varphi_{x_i}, f_e) = 1$ and $count(\varphi_{x_i}, q) = 0$ for all $q \in Q_i$, $q \neq f_e$. Let $\theta = \theta_{x_1} \cdot (\varepsilon, f_e, RESET, q_e) \cdot \theta_{x_2} \cdots \theta_{x_{n-1}} \cdot (\varepsilon, f_e, RESET, q_e) \cdot \theta_{x_n} \cdot (\varepsilon, f_e, RESET, f_r)$.

Then, there exist initial and accepting configurations $\gamma_0 = \{\phi\}$ and $\gamma_n = \{\varphi\}$ for A_r, such that $\gamma_0 \Rightarrow^\theta \gamma_n$, and $label(\theta) = label(\theta_{x_1}) \cdots label(\theta_{x_n}) = x$. Here, $\gamma_0 = \{\phi\}$ is the initial configuration of the accepting run of $A(e_1)$, where $count(\phi, q) = 0$ is added for every $q \notin Q_1$. Every added ε transition, goes from an accepting configuration for $A(e)$, to an initial configuration of $A(e)$. By the induction hypothesis, these accepting configurations only have one counter not equal to 1, the accept state, which is reset to zero by the $RESET$ transition. This guarantees a valid transition to the next initial configuration, which only has the count of its state, q_e, set to 1. Finally, the last ε transition of f_e to f_r, resets the count of f_e to zero and sets the count of f_r to 1, whereby φ satisfies the condition on its counters.

- If $r = e^{k..l}$, and $x \in L(r)$, then there are two possibilities. If $k = 0$, then x can be equal to ε. If $x = \varepsilon$, then there exist initial and accepting configurations for $A(r)$, such that for $\theta = (\varepsilon, q_r, RESET, f_{r2})$, $\gamma_0 \Rightarrow^\theta \gamma_n$ and $label(\theta) = \varepsilon$ and the count conditions on γ_n hold. If $x \neq \varepsilon$, then there exist $x_i \in L(e)$, $i = 1, \ldots, n$ and $k \leq n \leq l$, such that $x = x_1 \cdots x_n$. By induction, for every $i = 1, \ldots, n$, there exists a $\theta_{x_i} \in \delta_e^*$, and configurations $\gamma_{0,x_i} = \{\phi_{x_i}\}$ and $\gamma_{n,x_i} = \{\varphi_{x_i}\}$, such that $label(\theta_{x_i}) = x_i$, $\gamma_{0,x_i} \Rightarrow^{\theta_{x_i}} \gamma_{n,x_i}$, γ_{0,x_i} and γ_{n,x_i} are initial and accepting configurations of A_e respectively, and $count(\varphi_{x_i}, f_e) = 1$ and $count(\varphi_{x_i}, q) = 0$ for all $q \in Q_e$, $q \neq f_e$. Let $\theta = \theta_{x_1} \cdot (\varepsilon, f_e, RESET, f_{r1})(\varepsilon, f_{r1}, IN, q_e) \cdot \cdots \cdot (\varepsilon, f_e, RESET, f_{r1})(\varepsilon, f_{r1}, IN, q_e) \cdot \theta_{x_n} \cdot (\varepsilon, f_e, RESET, f_{r1})(\varepsilon, f_{r1}, RESET, f_{r2})$. Then, there exist initial and accepting configurations $\gamma_0 = \{\phi\}$ and $\gamma_n = \{\varphi\}$ for A_r, such that $\gamma_0 \Rightarrow^\theta \gamma_n$, and $label(\theta) = label(\theta_{x_1}) \cdots label(\theta_{x_n}) = x$. Here, $\gamma_0 = \{\phi\}$ is the initial configuration of the accepting run of $A(e_1)$, where $count(\phi, q) = 0$ is added for every $q \notin Q_1$. As for the star operator, we have added ε transitions to go from the accepting configurations of $A(e)$ to the initial configurations of $A(e)$. We have added two ε transitions between every word. The first goes from the accept state of $A(e)$ to f_{r1}, and f_{r1} goes to the start state of $A(e)$. Furthermore, the last ε transition is an IN-edge, which means that the counter of f_{r1} is not reset. For all the partial configurations ψ of the configurations in the computations of $A(e)$, we add $count(\psi, q) = 0$ for every state $q \notin Q_e$, $q \neq f_{r1}$. For f_{r1}, the count depends on the number of times we have passed f_{r1}, which is equal to the number of substrings read by $A(e)$. We also see that the fact that $count(\psi, f_{r1})$ is not equal to zero does not disturb the computation of $A(e)$, since $A(e)$ does not know f_{r1}. After reading the n substrings, we arrive at configuration $\gamma' = \{\varphi'\}$, with $state(\varphi') = f_{r1}$,

$count(\varphi', f_{r1}) = n$, and $count(\varphi', q) = 0$ for all other states q. Since, $k \leq n \leq l$, $mincount(f_{r1}) = k$, and $maxcount(f_{r2}) = k$, we can follow the RESET edge to f_{r2}. Here, we arrive at the accepting configuration $\gamma_n = \{\varphi\}$, for which $count(\varphi, f_{r2}) = 1$, and $count(\varphi, q) = 0$ for all other states q.

- If $r = e_1 \,\&\, e_2$ and $x \in L(r)$, then there exist $x_1 \in L(e_1)$ and $x_2 \in L(e_2)$ such that $x \in x_1 \,\&\, x_2$. By induction, for $i = 1, 2$, there exists a $\theta_{x_i} \in \delta^*_{e_i}$, and configurations $\gamma_{0,x_i} = \{\phi_{x_i}\}$ and $\gamma_{n,x_i} = \{\varphi_{x_i}\}$, such that $\gamma_{0,x_i} \Rightarrow^{\theta_{x_i}} \gamma_{n,x_i}$, γ_{0,x_i} and γ_{n,x_i} are initial and accepting configurations of A_{e_i} respectively, and $count(\varphi_{x_i}, f_i) = 1$ and $count(\varphi_{x_i}, q) = 0$ for all $q \in Q_i$, $q \neq f_i$.

  There must exist a $\theta_s \in \theta_1 \,\&\, \theta_2$ such that $label(\theta_s) = x$. Let $\theta = (\varepsilon, q_r, RESET, q_1, q_2)\theta_s(\varepsilon, f_1, f_2, RESET, f_r)$. Then there exist initial and accepting configurations $\gamma_0 = \{\phi\}$ and $\gamma_n = \{\varphi\}$ for A_r, such that $\gamma_0 \Rightarrow^{\theta} \gamma_n$, and $label(\theta) = label(\theta_s) = x$. The first transition $(\varepsilon, q_r, RESET, q_1, q_2)$ takes the initial configuration $\gamma_0 = \{\phi\}$ to $\gamma_1 = \{\phi_{x_1}, \phi_{x_2}\}$. The transitions of θ_{x_1} in θ_s take ϕ_{x_1} to φ_{x_1}, and the transitions of θ_{x_2} in θ_s take ϕ_{x_2} to φ_{x_2}. Hereby, we arrive in the configuration $\gamma' = \{\varphi_{x_1}, \varphi_{x_2}\}$. Finally, the last transition $(\varepsilon, f_1, f_2, RESET, f_r)$ takes us to $\gamma_n = \{\varphi\}$, where all conditions on the counters are respected.

□

Lemma 4.12. *For any string x, if $x \in L(A)$, then $x \in L(r)$.*

Proof. We proof this inductively on the structure of A:

- If $r = \emptyset$, then there does not exist an accepting run on A.
- If $r = \varepsilon$, then $\theta = (\varepsilon, q_r, RESET, q_f)$ is the only accepting run of A, and thus $L(A) = \{label(\theta)\} = \{\varepsilon\} = L(r)$.
- If $r = a$, then $\theta = (a, q_r, RESET, q_f)$ is the only accepting run of A, and thus $L(A) = \{label(\theta)\} = \{a\} = L(r)$.
- If $r = e_1 + e_2$, then, for $i = 1, 2$, every accepting run θ of A is equal to $(\varepsilon, q_r, RESET, q_i) \cdot \theta_x \cdot (\varepsilon, f_i, RESET, q_f)$, where θ_x is an accepting run for $A(e_i)$. Assume that $i = 1$, the proof for $i = 2$ is similar. By induction, we know that $label(\theta_x) \in L(e_1)$. Furthermore, we know that $L(e_1) \subseteq L(r)$, and $label(\theta_x) = label(\theta)$, and thus $label(\theta) \in L(r)$.

- If $r = e_1 \cdot e_2$, then every accepting run θ of A is equal to $\theta_{x_1} \cdot (\varepsilon, f_1, RESET, q_2) \cdot \theta_{x_2}$, where θ_{x_1} (resp. θ_{x_1}) is an accepting run for $A(e_1)$ (resp. $A(e_2)$). By induction, we know that $label(\theta_{x_1}) \in L(e_1)$, and $label(\theta_{x_2}) \in L(e_2)$. It follows that $label(\theta) = label(\theta_{x_1}) \cdot label(\theta_{x_2}) \in L(r)$.

- If $r = e^*$, then there are are two possibilities. The accepting run θ of A can be equal to $(\varepsilon, q_r, RESET, f_r)$. We know that $label(\theta) = \varepsilon \in L(r)$. The accepting run θ can also be equal to $\theta_{x_1} \cdot (\varepsilon, f_e, RESET, q_e) \cdot \theta_{x_2} \cdots \theta_{x_{n-1}} \cdot (\varepsilon, f_e, RESET, q_e) \cdot \theta_{x_n} \cdot (\varepsilon, f_e, RESET, f_r)$, for some $n \in \mathbb{N}$, where θ_{x_i} is an accepting run for $A(e)$, for $i = 1, \ldots, n$. Since $label(\theta) = label(\theta_{x_1}) \cdots label(\theta_{x_n})$, it follows, by induction, that $label(\theta) \in L(r)$.

- If $r = e^{k..l}$, then there are are two possibilities. If $k = 0$, the accepting run θ of A can be equal to $(\varepsilon, q_r, RESET, f_{r2})$. We know that $label(\theta) = \varepsilon \in L(r)$. The accepting run θ can also be equal to $\theta = \theta_{x_1} \cdot (\varepsilon, f_e, RESET, f_{r1})(\varepsilon, f_{r1}, IN, q_e) \cdot \cdots \cdot (\varepsilon, f_e, RESET, f_{r1})$ $(\varepsilon, f_{r1}, IN, q_e) \cdot \theta_{x_n} \cdot (\varepsilon, f_e, RESET, f_{r1})(\varepsilon, f_{r1}, RESET, f_{r2})$, for some $n \in \mathbb{N}$, $k \leq n \leq l$, where θ_{x_i} is an accepting run for $A(e)$, for $i = 1, \ldots, n$. Since $label(\theta) = label(\theta_{x_1}) \cdots label(\theta_{x_n})$, and $k \leq n \leq l$, it follows, by induction, that $label(\theta) \in L(r)$.

- If $r = e_1 \,\&\, e_2$, then every accepting run of θ of A must be equal to $(\varepsilon, q_r, RESET, q_1, q_2)\theta_s(\varepsilon, f_1, f_2, RESET, f_r)$, for which there exist θ_1, θ_2, such that for $i = 1, 2$, θ_i is an accepting run for $A(e_i)$, and $\theta_s \in \theta_1 \,\&\, \theta_2$. Therefore, $label(\theta_1) \in L(e_1)$, $label(\theta_2) \in L(e_2)$, and $label(\theta_s) \in label(\theta_1) \,\&\, label(\theta_2)$. It follows, by induction, that $label(\theta) = label(\theta_s) \in L(r)$.

□

Lemma 4.13. *The size of the ENFA $A(r)$, constructed by the previous algorithm on the RE$(\cdot, +, *, \#, \&)$-expression r, is polynomial in the size of r. Furthermore is the size of a configuration of $A(r)$ at any time polynomial in the size of r and $A(r)$.*

Proof. It is easy to see that $A(r)$ is polynomial in the size of $A(r)$. For every symbol or operator in r, we add a constant number of new states. Since the number of symbols and operators in r is a polynomial, it follows that the size of $A(r)$ is polynomial in the size of r.

For the second part of this lemma, we first note that the size of any partial configuration of $A(r)$ is polynomial in the size of $A(r)$. Indeed, it stores one state, and for every state it stores a counter in binary. These counters can

never become bigger than the maxcount value of their corresponding state, which are stored in $A(r)$. We thus have a polynomial number of counters of polynomial size.

Next, we show that the total number of partial configurations in a configuration of $A(r)$ can never be bigger than the number of shuffle operators in r plus 1. Every shuffle operator of r results in one split state and one corresponding merge state of $A(r)$. In the construction of $A(r)$ (Figure 4.1 and Figure 4.2), we see that every smaller automaton is always fully nested in its super automaton. It follows that when a split state of $A(r)$ splits a partial configuration in two partial configurations, it has to pass by its corresponding merge state before it can reach that split state again. In the worst case, $A(r)$ has *used* every split state and its configuration contains the number of shuffle operators in r plus 1 partial configurations. Before it can reach any split state again, it has to pass by its corresponding merge state, by which it reduces the number of partial configurations again. □

Lemma 4.14. *Suppose we have constructed an ENFA A for a RE*$(\cdot, +, *, \#, \&)$*-expression using the previously mentioned algorithm, then we can decide whether* $\varepsilon \in L(A)$ *in space polynomial in the size of A.*

Proof. The following non-deterministic algorithm decides whether $\varepsilon \in L(A)$:

1. Let γ be the initial configuration of B.
2. Guess whether to GOTO 3, or GOTO 4.
3. Guess a ε-transition $\theta \in \delta$, and check if there exists a configuration η, such that $\gamma \Rightarrow^{\theta} \eta$. If so, let $\gamma = \eta$, and GOTO 2, else REJECT.
4. If γ is an accepting configuration of B, ACCEPT, else, REJECT.

This algorithm follows a random number of ε-transitions. If $\varepsilon \in L(A)$, there is a run of this algorithm which reaches an accepting configuration and accepts. If this algorithm accepts, we have reached an accepting configuration by only following ε-transitions, and therefore $\varepsilon \in L(A)$.

By Lemma 4.13, we know that the sizes of γ and η are polynomial in the size of A. Therefore, this algorithm is in NPSPACE. Since NPSPACE = PSPACE, the lemma follows. □

Chapter 5

Tree languages

5.1 Trees

We can view an XML document as a finite tree with labels from a finite alphabet Σ. Therefore, we will abstract XML documents by Σ-trees. An XML document and its corresponding tree are shown in Figure 1.1 and Figure 1.5. Note that we leave out the actual data values, since these are not important for the structure of the tree.

The set of *unranked* Σ-*trees*, denoted by τ_Σ, is the smallest set of strings over Σ and the parenthesis symbols "(" and ")" such that, for $a \in \Sigma$ and $w \in \tau_\Sigma^*$, $a(w)$ is in τ_Σ. So, a tree is either ε (empty) or is of the form $a(t_1 \ldots t_n)$ where each t_i is a tree. In the tree $a(t_1 \ldots t_n)$, the subtrees $t_1 \ldots t_n$ are attached to the root labeled a. We write a rather than $a()$. Since there is no apriori bound on the number of children of a node in a Σ tree, these trees are *unranked*. We will also consider *binary trees*, trees where every internal node has exactly two children. For every $t \in \tau_\Sigma$, the *set of nodes* of t, denoted by $\mathrm{Dom}(t)$, is the set defined as follows:

- If $t = \varepsilon$, then $\mathrm{Dom}(t) = \emptyset$.
- If $t = a(t_1 \ldots t_n)$, where $a \in \Sigma$ and each $t_i \in \tau_\Sigma$, then $\mathrm{Dom}(t) = \{\varepsilon\} \cup \bigcup_{i=1}^{n} \{iu \mid u \in Dom(t_i)\}$.

The *label* of a node u in the tree $t = a(t_1 \ldots t_n)$, denoted by $lab^t(u)$, is defined as follows:

- If $u = \varepsilon$, then $lab^t(u) = a$.
- If $u = iu'$, then $lab^t(u) = lab^{t_i}(u')$.

We define the *depth* of a tree t, denoted by depth(t), as follows: if $t = \varepsilon$, then depth(t) $= 0$; and if $t = a(t_1 \cdots t_n)$ then depth(t) $= max\{depth(t_i) \mid i = 1, \ldots, n\} + 1$. A path in a tree t is a complete linear sequence of nodes from the root to the leaf. A word $w \in \Sigma^*$ is a Σ-path of a tree t if w is the sequence of labels of a path of t. In the sequel, whenever we say tree, we mean Σ-tree. A *tree language* is a set of trees. A tree language T is *homogeneous* if the root node of every tree in T has the same label.

5.2 Tree automata

As we did for string languages, we can also define automata for tree languages. These tree automata are more diverse than the string automata. We can distinguish between binary tree automata and unranked tree automata.

5.2.1 Binary tree automata

Binary tree automata can be deterministic or non-deterministic, and operate top down or bottom up. These differences give us the following four interesting tree automata.

Definition 5.1. *A* deterministic top down binary tree automaton (deterministic TD-BTA) *is a tuple* $A = (Q, \Sigma, \delta, q_0, (F_\sigma)_{\sigma \in \Sigma})$ *where*

- Q *is a finite set of states;*
- Σ *is a finite alphabet;*
- $\delta : Q \times \Sigma \rightarrow (Q \times Q)$ *is the transition function;*
- $q_0 \in Q$ *is the initial state; and*
- *for every* $\sigma \in \Sigma$*,* $F_\sigma \subseteq Q$ *is the finite set of accepting states for leaves with label* σ*.*

A *run* of a deterministic top down binary tree automata A on a binary tree t, is a function $\lambda : Dom(t) \rightarrow Q$, such that

- $\lambda(\varepsilon) = q_0$; and
- for every internal node $u \in Dom(t)$, with left child u_1 and right child u_2, $(\lambda(u_1), \lambda(u_2)) = \delta(\lambda(u), lab^t(u))$.

The run is *accepting* if for every leaf node u, $\lambda(u) \in F_{lab^t(u)}$. The *size* of a deterministic TD-BTA is $|Q| \times |\Sigma|$, the number of entries for the δ function.

Definition 5.2. *A* deterministic bottom up binary tree automaton (deterministic BU-BTA) *is a tuple* $A = (Q, \Sigma, \delta, (q_\sigma)_{\sigma\in\Sigma}, F)$ *where*

- Q *is a finite set of states;*
- Σ *is a finite alphabet;*
- $\delta : (Q \times Q) \times \Sigma \to Q$ *is the transition function;*
- *for every* $\sigma \in \Sigma$*,* q_σ *is the initial state for leaves with label* σ*; and*
- $F \subseteq Q$ *is the set of final states.*

A *run* of a deterministic bottom up binary tree automata A on a binary tree t, is a function $\lambda : Dom(t) \to Q$, such that

- $\lambda(u) = q_{lab^t(u)}$ for every leaf node u; and
- for every internal node $u \in Dom(t)$, with left child u_1 and right child u_2, $\lambda(u) = \delta(\lambda(u_1), \lambda(u_2), lab^t(u))$.

The run is *accepting* if $\lambda(\varepsilon) \in F$. The *size* of a deterministic BU-BTA is $|Q| \times |\Sigma|$.

Definition 5.3. *A* non-deterministic top down binary tree automaton (non-deterministic TD-BTA) *is a tuple* $A = (Q, \Sigma, \delta, I, (F_\sigma)_{\sigma\in\Sigma})$ *where*

- Q *is a finite set of states;*
- Σ *is a finite alphabet;*
- $\delta : Q \times \Sigma \to P(Q \times Q)$*, where* $P(Q \times Q)$ *is the powerset of* $Q \times Q$*, is the transition function;*
- $I \subseteq Q$ *is the set of initial states; and*
- *for every* $\sigma \in \Sigma$*,* F_σ *is the set of accepting states for leaves with label* σ*.*

A *run* of a non-deterministic top down binary tree automata A on a binary tree t, is a function $\lambda : Dom(t) \to Q$, such that

- $\lambda(\varepsilon) \in I$; and
- for every internal node $u \in Dom(t)$, with left child u_1 and right child u_2, $(\lambda(u_1), \lambda(u_2)) \in \delta(\lambda(u), lab^t(u))$.

The run is *accepting* if for every leaf node u, $\lambda(u) \in F_{lab^t(u)}$. The *size* of a non-deterministic TD-BTA is $|Q| \times |\Sigma|$, the number of entries for the δ function.

Definition 5.4. *A* non-deterministic bottom up binary tree automaton (non-deterministic BU-BTA) *is a tuple* $A = (Q, \Sigma, \delta, (I_\sigma)_{\sigma \in \Sigma}, F)$*, where*

- Q *is a finite set of states;*
- Σ *is a finite alphabet;*
- $\delta : (Q \times Q) \times \Sigma \rightarrow P(Q)$*, where* $P(Q)$ *is the powerset of* Q*, is the transition function;*
- *for every* $\sigma \in \Sigma$*,* I_σ *is the set of initial states for leaves with label* σ*; and*
- $F \subseteq Q$ *is the set of final states.*

A *run* of a non-deterministic bottom up binary tree automata A on a binary tree t, is a function $\lambda : Dom(t) \rightarrow Q$, such that

- $\lambda(u) \in I_{lab^t(u)}$ for every leaf node u; and
- for every internal node $u \in Dom(t)$, with left child u_1 and right child u_2, $\lambda(u) \in \delta(\lambda(u_1), \lambda(u_2), lab^t(u))$.

The run is *accepting* if $\lambda(\varepsilon) \in F$. The *size* of a non-deterministic BU-BTA is $|Q| \times |\Sigma|$.

5.2.2 Unranked tree automata

We have now defined the four possible (deterministic vs. non-deterministic and top down vs bottom up) automata for binary tree languages, and turn to unranked tree languages. Here, we will only define one variant, a non-deterministic bottom up automata. We do this because a deterministic unranked tree automaton would, because of its deterministic property, have to put a maximum rank on the trees it accepts. Therefore it would not be able to take full advantage of its unranked property. We do not define a top down automaton, because it would be too similar to our bottom up automaton. The only difference would be that the set of final states would become the set of initial states.

Definition 5.5. *An* unranked tree automaton (UTA) *is a tuple* $A = (Q, \Sigma, \delta, F)$*, where*

- *Q is a finite set of states;*
- *Σ is a finite alphabet;*
- *δ is a function $Q \times \Sigma \to Reg(Q)$ such that $\delta(q, a)$ is a regular expression over the alphabet Q for every $a \in \Sigma$ and $q \in Q$; and*
- *$F \subseteq Q$ is the set of final states.*

A *run* of A on a tree t is a function $\lambda : Dom(t) \to Q$ such that for every $v \in Dom(t)$ with n children, $\lambda(v1) \cdots \lambda(vn) \in \delta(\lambda(v), lab^t(v))$. Note that when v has no children, then the criterion reduces to $\varepsilon \in \delta(\lambda(v), lab^t(v))$. A run is *accepting* if $\lambda(\varepsilon) \in F$. The *size* of an UTA is $|Q| + |\Sigma|$ plus the sum of the sizes of all regular expressions occurring in δ.

A tree is accepted if there is an accepting run for it. The set of all accepted trees is denoted by $L(A)$.

Since we use regular expressions in our transition function, we can limit or extend the power of the UTA by specifying a class of regular expressions that are allowed in the UTA. If R is a class of regular expressions, then UTA(R) is the set of UTAs that only use the regular expressions in R in their transition function. For example, UTA(RE($+, \cdot, *$)) denotes the "standard" set of unranked tree automata.

5.2.3 Equivalence of tree automata

We have defined five different tree automata. In this section, we investigate the power of these different automata. Many of the proofs in this section are based on work by Thatcher ([Tha73]).

Theorem 5.6. *A set T of trees is recognizable by a non-deterministic BU-BTA if and only if T is recognizable by a deterministic BU-BTA.*

Proof. Of course, a deterministic automaton can never be more powerful than its non-deterministic counterpart. We show that we can simulate every non-deterministic BU-BTA $A = (Q, \Sigma, \delta, (I_\sigma)_{\sigma \in \Sigma}, F)$ by a deterministic BU-BTA $B = (Q', \Sigma, \delta', (q_\sigma)'_{\sigma \in \Sigma}, F')$, using a product construction.

The state set Q' of B is the powerset of Q, $P(Q)$. That is, B will at any node capture all states that A can be in at that node. The set of final states F' is equal to $\{C \mid C \cap F \neq \emptyset\}$. The automaton B accepts if at the root node, one of the states in its set of states is a final state for A. The transition function δ' must be created in such a way that given two sets of states C_1, C_2 and an alphabet symbol σ, it calculates all possible states the automaton A

can be in. That is, $\delta'(\sigma, C_1, C_2) = \bigcup_{q_i \in C_i} \delta(\sigma, q_1, q_2)$. Finally, the initial state for $\sigma \in \Sigma$, $q'_\sigma = I_\sigma$.

If A accepts a tree t, there exists an accepting run λ for A on t. Take the run λ' of B on t (there exists only one since B is deterministic). By construction, we know that for any node $u \in Dom(t)$, $\lambda(u) \in \lambda'(u)$. Furthermore, $\lambda(\varepsilon) \in F$ and thus, by construction, $\lambda'(\varepsilon) \in F'$.

Conversely, if B accepts a tree t, then its run λ' is an accepting run for B on t. Since λ' contains all possible runs of A, and since $\lambda'(\varepsilon) \cap F \neq \emptyset$ (λ' is an accepting run), we can construct an accepting run λ for A on t. □

Theorem 5.7. *A set T of trees is recognizable by a non-deterministic BU-BTA if and only if T is recognizable by a non-deterministic TD-BTA.*

Proof. In Lemma 5.8 we show how we can simulate a non-deterministic TD-BTA by a non-deterministic BU-BTA. In Lemma 5.9 we do the opposite. Together, these lemmas prove the theorem. □

Lemma 5.8. *A set T of trees is recognizable by a non-deterministic BU-BTA if T is recognizable by a non-deterministic TD-BTA.*

Proof. For any non-deterministic TD-BTA $A = (Q, \Sigma, \delta, I, (F_\sigma)_{\sigma \in \Sigma})$, we can construct an equivalent non-deterministic BU-TDA $B = (Q, \Sigma, \delta', (I_\sigma)'_{\sigma \in \Sigma}, F')$ as follows:

- $I'_\sigma = F_\sigma, \forall \sigma \in \Sigma$; and
- $F' = I$; and
- $\delta'(\sigma, q_1, q_2) = \{q \mid (q_1, q_2) \in \delta(\sigma, q)\}$

Let λ be an accepting run for A on a tree t , we claim that λ also is an accepting run for B on t. For all leaf nodes u with $lab^t(u) = \sigma$, $\lambda(u) \in F_\sigma$ and thus $\lambda(u) \in I'_\sigma$. Since δ and δ' are in fact the same functions, λ follows the transition function for δ' as it does for δ. This assures that λ is a run for B on t. Finally, $\lambda(\varepsilon) \in I$ and thus $\lambda(\varepsilon) \in F'$, which makes sure that λ is an accepting run for B on t.

The same reasoning holds for an accepting run λ for B on a tree t. It is also a run for A on t. It is an accepting run for A, since for every leaf node u, $\lambda(u) \in I'_\sigma$, and thus $\lambda(u) \in F_\sigma$. □

Lemma 5.9. *A set T of trees is recognizable by a non-deterministic TD-BTA if T is recognizable by a non-deterministic BU-BTA.*

Proof. For any non-deterministic BU-BTA $A = (Q, \Sigma, \delta, (I_\sigma)_{\sigma\in\Sigma}, F)$, we can construct an equivalent non-deterministic TD-BTA $B = (Q, \Sigma, \delta', I', (F_\sigma)'_{\sigma\in\Sigma})$ as follows:

- $F'_\sigma = I_\sigma, \forall\sigma \in \Sigma$; and
- $I' = F$; and
- $\delta'(\sigma, q) = \{(q_1, q_2) \mid q \in \delta(\sigma, q_1, q_2)\}$

Let λ be an accepting run for A on a tree t, we claim that λ also is an accepting run for B on t. Since $I' = F$ and $\lambda(\varepsilon) \in F$, it follows that $\lambda(\varepsilon) \in I'$. Since δ and δ' are in fact the same functions, λ follows the transition function for δ' as it does for δ. This assures that λ is a run for B on t. Finally, for all leaf nodes u with $lab^t(u) = \sigma$, $\lambda(u) \in F_\sigma$ and thus $\lambda(u) \in I'_\sigma$, which makes sure that λ is an accepting run for B on t.

The same reasoning holds for an accepting run λ for B on a tree t. It is also a run for A on t. It is an accepting run for A, since $\lambda(\varepsilon) \in I'$, and thus $\lambda(\varepsilon) \in F$. □

Theorems 5.6 and 5.7 show that deterministic BU-BTAs, non-deterministic BU-BTAs and non-deterministic TD-BTAs all recognize the same set of binary tree languages. One would expect that the non-deterministic TD-BTU are equally powerful. This, however, is not true.

Theorem 5.10. *The set of binary tree languages recognizable by a deterministic TD-BTA is a strict subset of the set of binary tree languages recognizable by a non-deterministic TD-BTA.*

Proof. Of course, the deterministic TD-BTA can never be more powerful than the non-deterministic TD-BTA. To prove that they are less powerful, we show that the tree language $\{a(ba), a(ab)\}$ can be recognized by a non-deterministic TD-BTA, but not by a deterministic TD-BTA.

To prove this formally, we will characterize the classes of languages recognized by deterministic TD-BTAs. Let Σ be a finite alphabet and R a regular subset of Σ^*. Define T_R to be the set of all trees t which have the property that every Σ-path of t is in R. For any regular R, T_R is recognizable by a TD-BTA.

Let $A = (Q, \Sigma, \delta, q_0, F)$ be a DFA that recognizes R. Let $B = (Q, \Sigma, \delta', (I_\sigma)'_{\sigma\in\Sigma}, F)$. Here, $\delta'(\sigma, q) = (\delta(\sigma, q), \delta(\sigma, q))$, where $\sigma \in \Sigma$ and $q \in Q$; and for every $\sigma \in \Sigma$, $(I_\sigma) = \{q_0\}$. The tree automaton B simulates the actions of A on every path of its tree. Since B has the same final states as A, B only accepts if every Σ-path is accepted by A. It follows that B recognizes T_R.

Example 5.11. *Let $\Sigma = \{a, b\}$ and R the subset of Σ^* defined by the regular expression a^*b, then T_R is the set of trees where the root and every internal node is labeled with a and every leaf node is labeled b.* □

This result shows the power of a deterministic TD-BTA. However, we are interested in its limitations. It turns out that its power does not reach much further than the power just stated.

A path in a tree t over an alphabet Σ can be represented by a string in $\{l, r\}^*$, where the l and r stand for the left and right branches in the tree. Furthermore, we add a Λ in the begin of the string to ensure that such a path has the same length as its corresponding path of labels. For example, Λlr is a path that goes left at the root and follows by going right to the leaf node. We represent a path with its labels as a pair (p, x), where p is a path over $\Lambda\{l, r\}^*$, x is a string of labels over Σ^*, and p and x have the same length. Using these label-path pairs, we can construct $\Delta_x = (\{\Lambda\} \times \Sigma) \cdot (\{l, r\} \times \Sigma)^*$, where $(\Lambda, \sigma_1)(\beta_2, \sigma_2) \cdots (\beta_k, \sigma_k)$ is equivalent to $(\Lambda\beta_2 \cdots \beta_k, \sigma_1 \cdots \sigma_k)$.

For a regular set R in Δ_Σ, let C_R be the set of trees t such that every label-path pair of t is in R. The exact power of the deterministic TD-BTAs is characterized by Magidor and Moran as follows

Lemma 5.12. *A set of trees T is recognizable by a deterministic TD-BTA if and only if $T = C_R$ for some regular subset R of Δ_Σ.*

Let's return to the tree language $\{a(ba), a(ab)\}$. Assume there exists a deterministic TD-BTA A which recognizes exactly $\{a(ba), a(ab)\}$. To accept the two given trees, the corresponding regular subset R of Δ_Σ must at least accept $(\Delta l, ab), (\Delta l, aa), (\Delta r, ab)$, and $(\Delta r, aa)$. It follows that A must accept $\{a(ba), a(ab), a(aa), a(bb)\}$. Since this set of trees is bigger than the original one, A can not exist.

The non-deterministic TD-BTA $B = (Q, \Sigma, \delta, I, (F_\sigma)_{\sigma\in\Sigma})$ which recognizes $\{a(ba), a(ab)\}$ is the following:

- $Q = \{q_s, q_a, q_b\}$; and
- $\Sigma = \{a, b\}$; and
- $\delta(q_s, a) = \{(q_a, q_b), (q_b, q_a)\}$; and
- $I = \{q_s\}$
- $F_a = \{q_a\}$, and $F_b = \{q_b\}$.

This shows that $\{a(ba), a(ab)\}$ is recognizable by non-deterministic TD-BTA and not by a deterministic TD-BTA, which concludes our proof. □

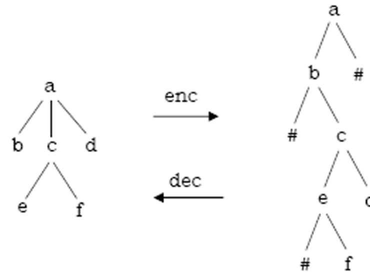

Figure 5.1: An unranked tree and its binary encoding.

It is hard to compare binary tree automata and unranked tree automata since they do not consider the same class of tree languages. However, we can encode every unranked tree into a binary tree. An example is shown in Figure 5.1. Here, the first child of a node u remains the first node of u but is explicitly encoded as its left child. The other children of u are right descendants of its first child. Whenever there is only one child, a # is inserted for the other child. Denote the encoding of a tree t by $enc(t)$ and the decoding of a tree t by $dec(t)$.

By using this encoding and decoding, the following has been proven:

Theorem 5.13. *([GKPS05],[Nev02],[Suc02])*

- *For every UTA($RE(+,\cdot,*)$) B there is a non-deterministic BU-BTA A such that $L(A) = \{enc(t) \mid t \in L(B)\}$. Moreover, A can be computed in polynomial time.*

- *For every non-deterministic BU-BTA A there is an UTA($RE(+,\cdot,*)$) B such that $L(B) = \{dec(t) \mid t \in L(A)\}$. Moreover, B can be computed in polynomial time.*

This theorem states that, in some sense, the BU-BTA and the UTA$(+,\cdot,*)$ have the same expressive power.

We can conclude that deterministic BU-BTAs, non-deterministic TD-BTAs, non-deterministic BU-BTAs, and UTA(RE$(+,\cdot,*)$)s all have the same expressive power. The deterministic TD-BTAs are less powerful than all the other automata.

5.3 XML schema languages

In this section, we present abstractions from XML schema languages, such as Document Type Definition, XML Schema, and Relax NG. We first give

an abstraction of a Document Type Definition.

Definition 5.14. *Let R be a class of regular expressions over Σ. A* DTD(R) *D is a tuple (d, s_d) where d is a function that maps Σ-symbols to expressions in R, and $s_d \in \Sigma$ is the start symbol.*

For convenience of notation, we denote (d, s_d) by d and leave the start symbol s_d implicit whenever this cannot give rise to confusion. A tree t satisfies d if

- $lab^t(\varepsilon) = s_d$; and
- for every $u \in Dom(t)$ with n children, $lab^t(u1) \cdots lab^t(un) \in L(d(lab^t(u)))$.

By $L(d)$ we denote the set of trees satisfying d. The *size* of a DTD D is $|\Sigma|$ plus the sum of the sizes of the regular expressions in d. For clarity, we write $d(a) = r$ as $a \to r$.

Example 5.15. *A DTD which defines the XML document of Figure 1.1 is shown in Figure 1.2. We can abstract from this DTD by the following DTD:*

$$\begin{array}{rcl} cd & \to & song^* \\ song & \to & title\ length\ singlesSold? \end{array}$$

XML Schema and Relax NG extend DTDs by a typing system. Therefore, Papakonstantinou and Vianu have extended this model with types ([PV00]).

Definition 5.16. *An* extended DTD (EDTD) *is a 5-tuple $D = (\Sigma, \Sigma', d, s_d, \mu)$, where Σ' is an alphabet of* types*, the tuple (d, s_d) is a DTD over Σ', and μ is a mapping from Σ' to Σ. Notice that μ can be extended to define a homomorphism on trees. A tree t then* satisfies *an extended DTD if $t = \mu(t')$ for some $t' \in L(d)$. Again, we denote by $L(D)$ the set of trees satisfying D.*

In the sequel, we also denote by μ the homomorphic extension of μ to strings, trees, or regular expressions. 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 $\mu(a_i) = a$. For a node u in a Σ-tree t, we say that a_i is a *type of u with respect to D* when

- there exists a Σ'-tree $t' \in L(D)$ such that $\mu(t') = t$; and
- $lab^{t'}(u) = a^i$.

The *size* of a EDTD D is $|\Sigma| + |\Sigma'|$ plus the sum of the sizes of the regular expressions in d. By EDTD(R) we denote the class of EDTDs in which the internal DTD is a DTD(R).

Example 5.17. *An XML Schema which defines the XML document in Figure 1.1, is shown in Figure 1.3. We create a type for a "single" song, and a "regular" song, and use these to define a cd. We also illustrate some extra features of XML Schema over DTD. The use of the* all *tag allows the child nodes of a song to occur in any order. By setting minoccurs to one and maxoccurs to twenty, we specify that a compact disc must have at least one song, and no more than twenty songs. The equivalent EDTD is the following:*

$$\begin{array}{rcl} cd & \rightarrow & (song^1 \ + \ song^2)^{1..20} \\ song^1 & \rightarrow & title \ \& \ length \ \& \ singlesSold \\ song^2 & \rightarrow & title \ \& \ length \end{array}$$

Here, $song^1$ *represents the* single *type songs, and* $song^2$ *represents the* regular *type songs.* □

We can express every XML schema or Relax NG document by an EDTD, but not every EDTD can be expressed by an XML Schema. That is because XML schema enforces some additional constraints. Relax NG does not have any constraints, and therefore is equally powerful as EDTDs.

One of these XML Schema constraints is Element Declarations Consistent (EDC). EDC says that in a regular expression, no elements with the same name, but with a different type can be used. Therefore, we describe a variant of EDTDs, single type EDTDs, which incorporates the EDC rule ([BMNS05],[MLMK05]).

Definition 5.18. *Let* $D = (\Sigma, \Sigma', d, s_d, \mu)$ *be an EDTD. A regular expression* r *over* Σ' *is* single-type *if there do not exist strings* $w_1 a^i v_1$ *and* $w_2 a^j v_2$ *in* $L(r)$*, with* $\mu(a^i) = \mu(a^j)$ *and* $i \neq j$*. We say that* D *is a* single-type EDTD (EDTDst) *when every regular expression* $d(a^i)$ *is single-type.*

By EDTD$^{st}(R)$, we denote the class of single-type EDTDs in EDTD(R).

Example 5.19. *If we look back at our example XML Schema (Figure 1.3), we see that it does not satisfy the EDC rule. Its corresponding EDTD also is not a single-type EDTD. To solve this, we can leave out the different types of the songs:*

$$\begin{array}{rcl} cd & \rightarrow & (song)^{1..20} \\ song & \rightarrow & title \ \& \ length \ \& \ singlesSold? \end{array}$$

Or, we can keep our types in tact by expanding our structure:

$$\begin{aligned} cd &\rightarrow singleSongs\ regularSongs \\ singleSongs &\rightarrow (song^1)^* \\ regularSongs &\rightarrow (song^2)^* \\ song^1 &\rightarrow title\ \&\ length\ \&\ singlesSold \\ song^2 &\rightarrow title\ \&\ length \end{aligned}$$

Note that this new EDTD is single-type, but does not define our original XML document (Figure 1.1) anymore. □

We have now defined DTDs, EDTDs, and EDTDst, which form an abstraction for DTD, Relax NG, and XML Schema, respectively. As mentioned in the introduction, our abstractions of DTD and XML Schema, do not take into account the *Deterministic Content Models* or equivalently *Unique Particle Attribution* constraints.

We say that a DTD (d, s) is *reduced* if for every symbol a that occurs in d, there exists a tree $t \in L(d)$ and a node $u \in Dom(t)$ such that $lab^t(u) = a$. For example, the DTD (d, a) where $\Sigma = \{a, b\}$, $d(a) = \varepsilon$ and $d(b) = b$ only defines the tree $a()$. Since b does not occur in this tree, the DTD is not reduced.

A DTD can be reduced in PTIME by the following algorithm ([MNS04]):

1. Compute the set $R \subseteq \Sigma$ of symbols that are reachable from the start symbol s of the DTD.
2. Compute in a bottom up manner the set of all symbols $S = \{a \mid L((d, a)) \neq \emptyset\}$.
3. Replace every occurrence of a symbol in $\Sigma \backslash (R \cap S)$ in a regular expression by $\emptyset$, and remove symbols in $\Sigma \backslash (R \cap S)$ from the DTD.

An EDTD or EDTDst is reduced if its DTD is reduced. In the sequel, we assume that all DTDs and EDTDs are reduced.

Theorem 5.20. *The class of tree languages definable by EDTDs is precisely the class of homogeneous tree languages definable by UTAs. Furthermore, given an EDTD D, we can construct an equivalent UTA, accepting homogeneous tree languages, in space logarithmic in the size of D; and given an UTA A, accepting homogeneous tree languages, we can construct an equivalent EDTD D, in space logarithmic in the size of A.*

Proof. We first show that for any EDTD, we can construct an equivalent UTA, accepting homogeneous tree languages, in logarithmic space(Lemma 5.21). Then, we prove the opposite (Lemma 5.23).

Lemma 5.21. *Given an EDTD D, we can construct an equivalent UTA A, accepting homogeneous tree languages, in space logarithmic in the size of D.*

Proof. Given an EDTD $D = (\Sigma_D, \Sigma'_D, d, s_d, \mu)$, we construct an UTA $A = (Q, \Sigma_A, \delta, F)$ such that $L(D) = L(A)$ as follows

- $Q = \Sigma'_D$; and
- $\Sigma_A = \Sigma_D$; and
- $\forall \sigma \in \Sigma'_D : \delta(\sigma, \mu(\sigma)) = d(\sigma)$; and
- $F = \{s_d\}$.

Proposition 5.22. *Let t be a tree satisfied by D. Let t' be the tree over Σ' such that t' is satisfied by d and $\mu(t') = t$. The function $\lambda : Dom(t) \to Q$, where for every node u of t, $\lambda(u) = lab^{t'}(u)$, defines an accepting run of A on t.*

Proof. We prove this proposition by induction on the depth of t. To do this, we need an extra notion of a run on a tree. We say that a run λ of A on t is *an accepting run up to depth k*, if $\lambda(\varepsilon) \in F$ and for every node u with $depth(u) \leq k$ and with n children, $\lambda(u1) \cdots \lambda(un) \in \delta(\lambda(u), lab^t(u))$. Obviously, a run λ of A on t is an accepting run if it is an accepting run up to $depth(t)$.

Basis: The node at depth 1 is the root node. We know that $\lambda(\varepsilon) = lab^{t'}(\varepsilon) = s_d$. Since $s_d \in F$, this defines an accepting run up to depth 1.

The induction step: Suppose that $\lambda(u) = lab^{t'}(u)$ defines an accepting run of A on t up to depth n, we prove that $\lambda(u) = lab^{t'}(u)$ defines an accepting run of A on t up to depth $n + 1$. For every node v of t' at depth n, we know that its children are in $d(v)$. Since we have defined $\delta(\sigma, \mu(\sigma))$ to be equal to $d(\sigma)$, we know that λ defines an accepting run of A on t up to depth $n + 1$. □

From this proposition, it follows that for every $t \in L(D)$, $t \in L(A)$. Therefore, $L(D) \subseteq L(A)$.

Conversely, if there exists an accepting run λ of A on a tree t, then t', where $lab^{t'}(u) = \lambda(u)$, is satisfied by d. This can be proven in an analogue manner as in Proposition 5.22. It follows that $L(A) \subseteq L(D)$.

It only remains to show that this construction can be done in logarithmic space. The construction of A consists in fact only of copying D, and can easily be done in logarithmic space. □

Lemma 5.23. *Given an UTA A, accepting homogeneous tree languages, we can construct an equivalent EDTD D, in space logarithmic in the size of A.*

Proof. Given an UTA $A = (Q, \Sigma_A, \delta, F)$ accepting homogeneous tree languages, we have to construct an EDTD $D = (\Sigma_D, \Sigma'_D, d, s_d, \mu)$ such that $L(A) \subseteq L(D)$.

We suppose that A has the following property: The set F is a singleton $\{q_f\}$ and $L(\delta(q_f, a))$ is empty for all but one $a \in \Sigma$. This symbol is the label of the root node of every tree accepted by A.

We show that for every UTA $A_1 = (Q_1, \Sigma_{A1}, \delta_1, F_1)$ accepting homogeneous tree languages, an equivalent UTA $A = (Q, \Sigma_A, \delta, F)$ with these properties exists. Since we are only considering homogeneous tree languages, we know that there is a unique root symbol, say s. Suppose that there exists an accept state q, for which $\delta(q, a) \neq \emptyset$, $a \neq s$, we know that this transition can never lead to an accepting run. We can thus replace these useless δ-values with the empty set.

We now create a new state, q_f, which will be the new and only accepting state. Let $\delta(q_f, s)$ be the concatenation of all $\delta(q, s)$-values, $q \in F_1$, separated by '+' tokens. The new value of F is $\{q_f\}$. We see that q_f replaces the old accept states as accept state and can never be used at another place in the tree. This guarantees that the new UTA is equivalent to the old UTA.

Now we know that A has the previously mentioned property, D can be constructed as follows:

- $\Sigma_D = \Sigma_A$;
- $\Sigma'_D = \{a^q \mid a \in \Sigma_A \text{ and } q \in Q\}$;
- $\mu(a^q) = a$;
- $d(a^q) = \delta(q, a)$ in which every state-symbol p is replaced by $(a_1^p + \cdots + a_n^p)$, assuming $\Sigma_A = \{a_1, \ldots, a_n\}$; and
- $s_d = s^{q_f}$, where q_f is the only accept state and s is the only symbol for which $\delta(q_f, a), a \in \Sigma$, is not empty.

Proposition 5.24. *Let t be a tree accepted by A, and $\lambda : Dom(t) \to Q$ the accepting run of A on t. Then, the tree t', with for any node $u \in Dom(t)$, $lab^{t'}(u) = lab^t(u)^{\lambda(u)}$, is satisfied by d, and $\mu(t') = t$.*

Proof. By construction, $\mu(t') = t$. We prove that t' is satisfied by d by induction on the depth of t'. We say that a tree t is *recognized up to depth* k by a DTD (d, s_d), if $lab^t(\varepsilon) = s_d$, and for every node u with $depth(u) \leq k$ and with n children, $lab^t(u1) \cdots lab^t(un) \in L(d(lab^t(u))$. Obviously, a tree t is recognized by (d, s_d) if it is recognized by (d, s_d) up to $depth(t)$.

Basis: $lab^{t'}(\varepsilon) = lab^t(\varepsilon)^{\lambda(\varepsilon)} = s^{q_f}$, which is the start symbol of d. Therefore, d recognizes t' up to depth 1.

The induction step: Suppose that t' is recognized by d up to depth n, we prove that t' is recognized by d up to depth $n + 1$. For every node v of t' at depth n, we know that its children are in $\delta(\lambda(v), lab^t(v))$. If v has n children, we can write these as $\lambda(v1) \cdots \lambda(vn)$. Since we have defined $d(a^q)$ to be $\delta(q, a)$, where every state-symbol p is replaced by $(a_1^p + \cdots + a_n^p)$, we know that $lab^t(v1)^{\lambda(v1)} \cdots lab^t(vn)^{\lambda(vn)} \in L(d(lab^t(v)^{\lambda(v)}))$. □

From this proposition, it follows that for every $t \in L(A)$, $t \in L(D)$. Therefore, $L(A) \subseteq L(D)$.

Conversely, suppose that a tree t is satisfied by D. Then there exists a tree t' such that $lab^{t'}(u) = lab^t(u)^{type(u)}$, for every node u of t, and t' is satisfied by d. The accepting run of A on t is λ, where $\lambda(u) = type(u)$, for every node of t. This can be proven in an analogue manner as in Proposition 5.24. It follows that $L(D) \subseteq L(A)$.

It only remains to show that this construction can be done in logarithmic space. We have to construct a type for every symbol-state combination, and we have to alter the transition function of A. This can all be done in logarithmic space. □

This concludes the proof of Theorem 5.20. □

5.4 Decision problems

Definition 5.25. *Let* `M` *be a subclass of the class of regular expressions, tree automata, DTDs, EDTDs, and EDTD*st*s.*

- *Inclusion for* `M`*: Given two expressions, automata, or schemas $d, d' \in$* `M`*, is $L(r) \subseteq L(r')$?*

- *Equivalence for* `M`*: Given two expressions, automata, or schemas $d, d' \in$* `M`*, is $L(r) = L(r')$?*

- *Intersection (non-emptiness) for* `M`*: Given an arbitrary number of expressions, automata, or schemas $d_1, \ldots, d_n \in$* `M`*, is $\bigcap_{i=1}^n L(d_i) \neq \emptyset$?*

Let `R` *be a subclass of the class of regular expressions.*

- *Universality for* `R`*: Given an expression $r \in$* `R`*, is $L(r) = L(\Sigma^*)$?*

The inclusion, equivalence and intersection problem are the fundamental problems studied in this work. The universality problem is sometimes useful to prove hardness for the inclusion and equivalence problem.

In this work, we are mostly interested in the complexity of these problems for DTDs, EDTDs, and, EDTDsts, because of their correspondence to schema languages used in practice. However, the complexities of these problems are closely related to these of regular expressions, and unranked tree automata.

Theorem 5.26. *([MNS04]) Let* `R` *be be a class of regular expressions and* `C` *be a complexity class which is closed under positive reductions. Then the following are equivalent:*

- *Inclusion for* `R` *is in* `C`*.*
- *Inclusion for DTD(*`R`*) is in* `C`*.*
- *Inclusion for EDTDst(*`R`*) is in* `C`*.*

The corresponding statement holds for the equivalence problem.

Theorem 5.27. *([MNS04]) Let* `R` *be be a class of regular expressions and* `C` *be a complexity class which is closed under positive reductions. Then the following are equivalent:*

- *Intersection for* `R` *is in* `C`*.*
- *Intersection for DTD(*`R`*) is in* `C`*.*

These theorems show that by establishing the exact complexities for regular expressions, we know the complexity of these problems for DTDs and EDTDsts. The only problem that does not fall under this classification is the intersection problem for EDTDsts. Since every EDTD can be simulated by an UTA, the upper bound for the intersection problem for UTAs also gives us an upper bound for the intersection problem for EDTDsts. Furthermore, it follows that the complexity of the decision problems for UTAs immediately carry over to EDTDs.

Because of this close correspondence between regular expressions and unranked tree automata on one hand, and DTDs, EDTDsts, and EDTDs on the other hand, we will study the decision problems for various interesting subclasses of regular expressions and unranked tree automata. Regular expressions are studied in chapter 6, unranked tree automata in chapter 7.

Chapter 6

Complexity of regular expressions

6.1 Regular expressions

In this section, we investigate the complexity of the equivalence, inclusion and intersection problem for regular expressions. We do this for a number of interesting subclasses of regular expressions. The results are summarized in Table 6.1.

	equivalence	inclusion	intersection
$\mathrm{RE}(+,\cdot,*)$	PSPACE [SM73] (6.1)	PSPACE [SM73] (6.1)	PSPACE [Koz77] (6.7)
$\mathrm{RE}(+,\cdot,*,\#)$	EXPSPACE (6.2)	EXPSPACE (6.2)	PSPACE (6.8)
$\mathrm{RE}(+,\cdot,*,\&)$	EXPSPACE [MS94] (6.2)	EXPSPACE [MS94] (6.2)	PSPACE (6.8)
$\mathrm{RE}(+,\cdot,*,\#,\&)$	EXPSPACE (6.2)	EXPSPACE (6.2)	PSPACE (6.8)

Table 6.1: Complexity classes for problems with regular expressions. All complexities are completeness results. For results already known in the literature, the reference is added. The theorem numbers are given in brackets.

We see that the results for the equivalence and inclusion problem are the same. For the "standard" set of regular expressions $\mathrm{RE}(+,\cdot,*)$, these problems are in PSPACE. When we add the numerical occurrence operator or the shuffle operator, we immediately go one exponential higher, to EXPSPACE. However, when we add both operators, we remain in EXPSPACE.

The results for the intersection problem came as a surprise. For the standard set of regular expressions, the intersection problem also is in PSPACE. One would expect that the addition of one of the two extra operators would take it to EXPSPACE. This, however, is not the case. We can even add both

operators and still remain in PSPACE.

Note that PSPACE and EXPSPACE are closed under complement and therefore the same completeness results hold for the complement of the equivalence, inclusion and intersection problem.

6.1.1 Equivalence and inclusion

Theorem 6.1. *([SM73]) The equivalence, and inclusion problem for RE$(+, \cdot, *)$ are PSPACE-complete.*

Theorem 6.2. *The equivalence and inclusion problem for RE$(+, \cdot, *, \#, \&)$ are EXPSPACE-complete. These problems remain EXPSPACE-hard for RE$(+, \cdot, *, \#)$ and RE$(+, \cdot, *, \&)$.*

Proof. We prove this theorem in a series of lemmas. First, we prove that the equivalence and inclusion problem for RE$(+, \cdot, *, \#, \&)$ are in EXPSPACE (Lemma 6.3). In Lemma 6.4, we prove that the equivalence and inclusion problem for RE$(+, \cdot, *, \#)$ are EXPSPACE-hard. It has already been shown that the equivalence problem for RE$(+, \cdot, *, \&)$ is EXPSPACE-complete ([MS94], Lemma 6.5). We use this to show that the inclusion problem for RE$(+, \cdot, *, \&)$ is EXPSPACE-hard. Together, these lemmas prove the theorem.

Lemma 6.3. *The equivalence and inclusion problem for RE$(+, \cdot, *, \#, \&)$ are in EXPSPACE.*

Proof. We give a non-deterministic EXPSPACE algorithm that decides the complement of the inclusion problem for RE$(+, \cdot, *, \#, \&)$. Since NEXPSACE = EXPSPACE, and EXPSPACE is closed under complement, it follows that the inclusion problem is in EXPSPACE. Finally, we use the algorithm for the complement of the inclusion problem to show that the equivalence problem for RE$(+, \cdot, *, \#, \&)$ is in EXPSPACE.

On input RE$(+, \cdot, *, \#, \&)$-expressions r_1, r_2, we have to decide whether $L(r_1) \not\subseteq L(r_2)$. In order to do this, we construct extended NFAs A_1, and A_2 such that $L(A_1) = L(r_1)$, and $L(A_2) = L(r_2)$, using the construction described in section 4.2.

We will now try to guess a string, character by character, that is accepted by A_1, but not by A_2. Therefore, we maintain at any time a set of configurations C_1 (resp. C_2) which contains all configurations that A_1 (resp. A_2) can be in after reading the current string.

We define the ε-closure of C_i, $i = 1, 2$, as C_i expanded with the set of all configurations reachable from configurations in C_i using only ε-transitions in A_i. The ε-closure of C_i can be computed by searching all ε-transitions in A_i

that can be followed by one of the configurations in C_i. Then, we add all new configurations that are reached by following these transitions to C_i. We repeat this until C_i does not change anymore.

We can now describe the algorithm:

1. On input r_1, r_2, construct ENFAs $A_1 = (Q_1, \Sigma, q_1, f_1, \delta_1)$, $A_2 = (Q_2, \Sigma, q_2, f_2, \delta_2)$ such that $L(A_1) = L(r_1)$, and $L(A_2) = L(r_2)$.

2. For $i = 1, 2$, let $C_i = \{\gamma_i\}$, where γ_i is the initial configuration of A_i.

3. For $i = 1, 2$, let $\overline{C_i}$ be the ε-closure of C_i, and let $C_i = \overline{C_i}$.

4. If C_1 contains an accepting configuration, and C_2 does not, ACCEPT.

5. Guess the next symbol $a \in \Sigma$

6. For $i = 1, 2$, let $W_i = \{(a, p, c, q) \in \delta_i \mid p, q \in Q_i, c \in \{IN, RESET\}\} \cup \{(a, p, RESET, q, q') \in \delta_i \mid p, q, q' \in Q_i\} \cup \{(a, p, p', RESET, q) \in \delta_i \mid p, p', q \in Q_i\}$. That is, W_i is the set of all transitions of A_i that follow an a.

7. For $i = 1, 2$, let $C_i = \{\gamma \mid \gamma_i \Rightarrow^w \gamma, \gamma_i \in C_i, w \in W_i\}$.

8. GOTO 3.

This algorithm will non-deterministically try to guess a string, character by character, that is recognized by r_1 but not by r_2. In step 1, we construct extended NFAs corresponding to the regular expressions r_1 and r_2, and we initialize the sets of configurations C_1 and C_2 by the initial configurations of A_1 and A_2 (step 2). We compute the ε-closure of C_1 and C_2 to find all configurations A_1 and A_2 can be in without reading a symbol (step 3). Then, we check if C_1 contains an accepting configuration (A_1 can accept), and C_2 does not (A_2 can not accept). If that is the case, we have guessed a string that is accepted by A_1 and not by A_2. At this point, that string is ε, but we return here after guessing every following symbol. In step 5, we guess the first symbol a of our string. Then, we compute the sets of all possible transitions that follow the symbol a, W_i, and use this to recompute the C_i (steps 6 and 7). Then, we go back to step 3, where we again compute the ε-closure of the C_i, and check if the string is accepted by A_1, and not by A_2.

Suppose, $L(r_1) \not\subseteq L(r_2)$, then there must exist a string x, such that $x \in L(r_1)$ and $x \notin L(r_2)$. There exists a run of our algorithm that guesses x, and accepts. Suppose that our algorithm accepts. That means that it has guessed a string x, such that $x \in L(r_1)$ and $x \notin L(r_2)$. It follows that $L(r_1) \not\subseteq L(r_2)$.

Finally, we show that this algorithm operates in exponential space. For $i = 1, 2$, we know that the size of the A_1 and A_2 is polynomial in the size of r_1 and r_2, and that the size of the configurations of A_i is polynomial in the size of A_i and r_i (Lemma 4.13). If the size of the configurations is bounded by a polynomial $f(n)$, then we know that there can not be more than $2^{f(n)}$ different configurations. Therefore, the sets $C_1, \overline{C_1}, C_2$, and $\overline{C_2}$ can maximally contain exponentially many configurations of polynomial size, and thus have an exponential size. The symbol a, and the sets W_1 and W_2 can all be stored in polynomial space. The computation of these sets in the different steps, can easily be done in exponential space.

We now have an algorithm to decide the complement of the inclusion problem. An algorithm for the complement of the equivalence problem is the following: on input RE($+, \cdot, *, \#, \&$)-expressions r_1, r_2, guess whether to compute whether $L(r_1) \not\subseteq L(r_2)$ or $L(r_2) \not\subseteq L(r_1)$. Again, since NEXPSPACE = EXPSPACE, and EXPSPACE is closed under complement, it follows that the equivalence problem for RE($+, \cdot, *, \#, \&$) is in EXPSPACE. □

Lemma 6.4. *The equivalence and inclusion problem for RE($+, \cdot, *, \#$) are EXPSPACE-hard.*

Proof. To show that the equivalence problem for RE($+, \cdot, *, \#$) is EXPSPACE-hard, we do a reduction from the 2^n-corridor tiling problem, which is EXPSPACE-complete (Theorem 3.5). On input a tiling system $\tau = \langle O, o_{bot}, o_{top}, n \rangle$, we construct two regular expressions r_1 and r_2 such that $L(r_1) = L(r_2)$ if and only if there exists a 2^n-corridor tiling for τ.

To this end, we define a string representation for a tiling. Each row of the tiling, denoted by R_i, is represented by the tiles it consists of, for example "$o_0 o_2 o_3 o_1$" for a row of length four. An entire tiling with m rows is represented by its rows, separated by #-tokens. This gives us $\#R_0\# \cdots \#R_m\#$, R_0 being the bottom row and R_m being the top row.

Now we define r_1 and r_2. Let $\Sigma = O \cup \{\#\}$, and $r_1 = \Sigma^*$. We define r_2 as the regular expression that captures all possible strings except those that represent the encoding of a correct 2^n-corridor tiling for τ. If r_2 captures all possible strings, and thus is equal to Σ^*, there does not exist a 2^n-corridor tiling for τ. Conversely, if $L(r_1) \neq L(r_2)$, there does exist a 2^n-corridor tiling for τ.

We give a regular expression for every possible error in a string. We define r_2 as the disjunction of the following expressions:

1. The string does not start or end with a $\#$: $O\Sigma^* + \Sigma^* O$.

2. Between two successive #-tokens, there are more or less than 2^n tiles: $\Sigma^*\#O^{0..2^n-1}\#\Sigma^* + \Sigma^*\#O^{2^n+1..2^n+1}O^*\#\Sigma^*$.

3. Two tiles are horizontally adjacent, but this is not permitted by the tiling system: $\forall o_1, o_2 \in O : (o_1, o_2) \notin H \rightarrow \Sigma^* o_1 o_2 \Sigma^*$.

4. Two two tiles are vertically adjacent, but this is not permitted by the tiling system: $\forall o_1, o_2 \in O : (o_1, o_2) \notin V \rightarrow \Sigma^* o_1 \Sigma^{2^n..2^n} o_2 \Sigma^*$.

5. The tile at position (1,1) is not equal to o_{bot}, let $O_b = O\backslash\{o_{bot}\}$: $\#O_b\Sigma^*$.

6. The tile at position (m,1) is not equal to o_{top}, let $O_t = O\backslash\{o_{top}\}$: $\Sigma^*\#O_tO^*\#$.

By construction, we see that every string that is captured by one of the above expressions, does not express a correct 2^n-corridor tiling for τ. We have to prove that if a string is not captured by one of these expressions, it indeed is a valid encoding of a 2^n-corridor tiling for τ. Since the string does not match the regular expressions in items 1 and 2 it already is a tiling. That is, a sequence of rows of tiles of length 2^n. The regular expressions of items 3 and 4 assure that all tiles that are horizontally and vertically adjacent, have the permission of the tile system to border. Items 5 and 6 assure that the tiles o_{bot} and o_{top} are positioned at the right places. We can conclude that a string which is not captured by one of the previous regular expressions, represents a correct 2^n-corridor tiling.

It only remains to show that this reduction can be done in logarithmic space. The regular expression in item 1 has to list all members of O. This can be done by keeping a pointer to the set O, which requires only logarithmic space with respect to the size of O. In item 2 we have to generate the numbers $2^n - 1$ and $2^n + 1$. In binary these numbers are of the form $11 \cdots 1$, with n ones, and $100 \cdots 01$, with $n-1$ zeros. It is easy to generate these numbers by writing n in binary on the worktape and thus only using logarithmic space. In item 3, we have to keep two pointers to O, which both require logarithmic space. Checking if a tuple of two tiles is in H is easy. Item 4 is a combination of items 3 and 2. We also have to keep two pointers to O, check if a tuple of tiles is in V and generate the number 2^n. Again, 2^n written in binary is $100 \cdots 00$ with n zeros and can be generated in logarithmic space by writing n in binary on the work tape. The regular expressions in items 5 and 6 impose no new problems.

The proof for the EXPSPACE-hardness of the inclusion problem is completely analogue. We generate the same regular expressions r_1 and r_2. Since r_1 is equal to Σ^*, $r_1 \subseteq r_2$ if and only if $r_1 = r_2$. □

For RE$(+, \cdot, *, \&)$, it has been shown in [MS94] that the complement of the equivalence problem and the complement of the universality problem are EXPSPACE-complete. Since EXPSPACE is closed under complement, the following easily follows:

Lemma 6.5. *The equivalence and universality problem for RE$(+, \cdot, *, \&)$ are EXPSPACE-complete.*

Lemma 6.6. *The inclusion problem for RE$(+, \cdot, *, \&)$ is EXPSPACE-hard.*

Proof. We reduce the universality problem for RE$(+, \cdot, *, \&)$, which is EXPSPACE-complete (Lemma 6.5), to the inclusion problem for RE$(+, \cdot, *, \&)$. Given a RE$(+, \cdot, *, \&)$-expression r, output $r_1 = \Sigma^*$ and $r_2 = r$. Since $r = \Sigma^*$ if and only if $\Sigma^* \subseteq r$, this is a valid reduction. This can of course be done in logarithmic space. □

This concludes the proof of Theorem 6.2. □

6.1.2 Intersection

Theorem 6.7. *([Koz77]) The intersection problem for RE$(+, \cdot, *)$ is PSPACE-complete.*

Theorem 6.8. *The intersection problem for RE$(+, \cdot, *, \#, \&)$ is PSPACE-complete. It remains PSPACE-hard for RE$(+, \cdot, *, \#)$ and RE$(+, \cdot, *, \&)$.*

Proof. We already know that the intersection problem for RE$(+, \cdot, *)$ is PSPACE-hard (Theorem 6.1). We only have to show that the intersection problem for RE$(+, \cdot, *, \#, \&)$ is in EXPSPACE (Lemma 6.9). The Theorem follows.

Lemma 6.9. *The intersection problem for RE$(+, \cdot, *, \#, \&)$ is in PSPACE.*

Proof. On input RE$(+, \cdot, *, \#, \&)$-expressions $r_1, \ldots, r_n$, we have to decide whether $\bigcap_{i=1}^n L(r_i) \neq \emptyset$. In order to do this, we start by constructing extended NFAs $A_1, \ldots A_n$, such that for every i, $L(A_i) = L(r_i)$. Then, we try to guess a string, character by character, that is accepted by every A_i. In the analogues algorithm for the equivalence and inclusion problem (Lemma 6.3), we always maintained a set of all configurations the A_i could be in after reading the current string. We had to do this because we also had to make sure that a string is not accepted by an automaton. This is not required in the intersection problem. Therefore, we maintain at any time just one of the possible configurations γ_i, A_i can be in. Of course, these configurations γ_i can contain more than one partial configuration.

The algorithm goes as follows:

1. On input $r_1, \ldots, r_n$, construct ENFAs $A_1, \ldots A_n$, $A_i = (Q_i, \Sigma, q_i, f_i, \delta_i)$, such that $L(A_i) = L(r_i)$.
2. For $i = 1, \ldots, n$, let γ_i be the initial configuration of A_i.
3. For $i = 1, \ldots, n$, guess whether to follow an ε-transition of A_i and update γ_i accordingly.
4. Guess whether to repeat step 3 (GOTO 3) or continue with the algorithm (GOTO 5).
5. If, for every i, γ_i is an accepting configuration for A_i, ACCEPT.
6. Guess the next symbol $a \in \Sigma$.
7. For $i = 1, \ldots, n$, guess a transition $w_i \in \delta_i$. If, for any i, the first element of w_i (the symbol read) is not equal to a, or there does not exist a configuration η_i such that $\gamma_i \Rightarrow^{w_i} \eta_i$, REJECT. Else, for every i, let $\gamma_i = \eta_i$.
8. GOTO 3.

This algorithm will non-deterministically try to guess a string, character by character, that is recognized by every r_i, $i = 1, \ldots, n$. In step 1, we construct extended NFAs corresponding to the regular expressions $r_1, \ldots, r_n$, and we initialize the configurations of the A_i, by their initial configurations (step 2). In step 3 and 4, we allow the automata to follow an arbitrary number of ε-transitions and update the configurations accordingly. If all γ_i are accepting configurations, this means that we have guessed a string that is accepted by every r_i and that we have guessed a valid accepting run of every A_i on this string (step 5). At this point, that string is ε, but we return here after guessing every following symbol. In step 6, we guess the first symbol a of our string. Then, we guess a transition that the automata will follow. Of course, this transition must follow an a, and be a valid transition given the current configuration. If that is the case, we update the configurations accordingly (step 7). Then, we return to step 3, where we again allow the automata to follow ε-transitions, and can check whether the current string is accepted by every automaton.

Suppose, $\bigcap_{i=1}^{n} L(r_i) \neq \emptyset$, then there must exist a string x, such that $x \in L(r_i)$, for every i. There exists a run of our algorithm that guesses x, guesses a valid accepting run for every A_i on x, and accepts. Suppose that our algorithm accepts. That means that it has guessed a string x, such that $x \in L(r_i)$, for every i. It follows that $\bigcap_{i=1}^{n} L(r_i) \neq \emptyset$.

Finally, we show that this algorithm operates in polynomial space. By Lemma 4.13, we know that the size of the constructed A_is is polynomial in the size of the r_is, and that the size of the configurations of A_i is polynomial in the size of A_i and r_i. Therefore, the configurations γ_i are of polynomial size. The symbol a can of course stored in polynomial space. Guessing transitions, checking whether transitions are valid, and updating the current configurations can all be done in polynomial space.

□

This concludes the proof of Theorem 6.8. □

6.2 Chain regular expressions

As mentioned in the introduction, we also study a simpler class of regular expressions, CHAREs. The definition of these CHAREs can be found in Section 4.1. We present some of the results obtained in [MNS04] on these CHAREs, and investigate what happens when we add the numerical occurrence indicator to the class of CHAREs. The results of this section are summarized in Table 6.2.

	equivalence	inclusion	intersection
CHARE($S\backslash\{\#\}$)	in PSPACE * (6.10)	PSPACE * (6.10)	PSPACE * (6.10)
CHARE(S)	in EXPSPACE (6.12)	EXPSPACE (6.11)	PSPACE (6.13)
CHARE($a, a?$)	in PTIME * (6.14)	coNP *(6.14)	NP *(6.8)
CHARE($a, a*$)	in PTIME * (6.14)	coNP *(6.14)	NP *(6.8)
CHARE($a, a?, a\#$)	in PTIME (6.15)	coNP-hard, in EXPSPACE (6.20)	NP (6.17)
CHARE($a, a\#^{>0}$)	in PTIME (6.21)	in PTIME (6.21)	in PTIME (6.21)

Table 6.2: Complexity classes for problems with CHAREs. All complexities are completeness results, unless mentioned otherwise. Results marked with a * are due to Martens et. al ([MNS04]). The theorem numbers are given in brackets.

For the class of CHAREs without the numerical occurrence indicator, CHARE($S\backslash\{\#\}$), the complexities are almost the same as the complexities of its *parent* class RE($+, \cdot, *$) (Theorem 6.1,6.7). That is, the results are the same for the inclusion and intersection problem, but the exact complexity of the equivalence problem has not yet been established. Of course, it is in $PSPACE$, but it is possible that it is more tractable.

When we add the numerical occurrence indicator, CHARE(S), we get the same result. The inclusion problem is EXPSPACE-complete, and the intersection problem is PSPACE-complete. These complexities are the same as the complexities of its *parent* class RE($+, \cdot, *, \#$) (Theorems 6.2 and 6.8). The complexity of the equivalence problem for CHARE(S) remains open. We know that is in EXPSPACE, but it is again possible that it is more tractable.

We study these CHARE expressions because we want to find subsets of the set of regular expressions that are used in practice and for which our problems become tractable. Therefore, we also consider simple subsets of CHARE-expressions where no disjunctions are allowed. One would expect the corresponding decision problems to be tractable. However, most problems quickly turn out to be NP or coNP-complete. The inclusion and intersection problem for CHARE($a, a?$) and CHARE($a, a*$) are coNP and NP-complete. The equivalence problem for these classes is in PTIME.

When we add the numerical occurrence indicator to the CHARE($a, a?$)-expressions, the complexities seem to remain the same. Intersection for CHARE($a, a?, a\#$) is NP-complete and the equivalence problem remains in PTIME. For the inclusion problem, we only know that it is also coNP-hard.

The equivalence problem for CHARE($a, a?, a\#$) is in PTIME, but can be we also find a subset of the CHARE-expressions with numerical occurrence indicators for which the inclusion and intersection problem is tractable? We can, but we have to restrict the power of the numerical occurrence indicator. Let CHARE($a, a\#^{>0}$) be the set of regular expression in which only factors of the form a ($a \in \Sigma$) or $a^{i..j}$ ($a \in \Sigma, i, j \in \mathbb{N}$ and $i > 0$) are allowed. The only difference with CHARE($a, a\#$) is that we do not allow the lower bounds of the numerical occurrence indicators to be zero. For CHARE($a, a\#^{>0}$), we show that the equivalence, inclusion and intersection problem are in PTIME.

Theorem 6.10. *([MNS04])*

- *The inclusion problem for CHARE($S\backslash\{\#\}$) is PSPACE-complete.*
- *The equivalence problem for CHARE($S\backslash\{\#\}$) is in PSPACE.*
- *The intersection problem for CHARE($S\backslash\{\#\}$) is PSPACE-complete.*

Theorem 6.11. *The inclusion problem for CHARE(S) is EXPSPACE-complete.*

Proof. This proof is essentially the same as the proof for the PSPACE-completeness of the inclusion problem for CHARE($S\backslash\{\#\}$) ([MNS04]). We only use the 2^n-corridor tiling problem instead of the normal corridor tiling problem and the definitions of the regular expressions are somewhat different.

Since the inclusion problem for RE($+, \cdot, *, \#$) is in EXPSPACE (Theorem 6.2), and the regular expression of CHARE(S) are a strict subset of these in RE($+, \cdot, *, \#$), it follows that the inclusion problem for CHARE(S) is in EXPSPACE. To show that it is EXPSPACE-hard, we do a reduction from the 2^n-corridor tiling problem, which is EXPSPACE-complete (Theorem 3.5). Given a tiling system $\tau = \langle O, o_{bot}, o_{top}, n\rangle$, we construct regular expressions r_1 and r_2 in such that $L(r_1) \subseteq L(r_2)$ if and only if there exists no 2^n-corridor tiling for τ.

We use the same string representation to encode a tiling as in the proof for the EXPSPACE-hardness of the inclusion problem for RE($+, \cdot, *, \#$) (Lemma 6.4). That is, $\#R_0\#R_1\#\cdots\#R_m\#$, in which each R_i represents a row and is of the form O^{2^n}, and R_0 is the bottom row and R_m is the top row. We define some collections of symbols: $O_\# = O \cup \{\#\}$ and $O_{\$,\#} = O \cup \{\$, \#\}$. The following regular expressions detect strings that do not encode a correct 2^n-corridor tiling for τ:

1. Between two successive $\#$-tokens, there may not be less than 2^n tiles: $O_\#^*\#O^{0..2^n-1}\#O_\#^*$.

2. Between two successive $\#$-tokens, there may not be more than 2^n tiles: $O_\#^*\#O^{2^n+1..2^n+1}O^*\#O_\#^*$.

3. Two tiles are horizontally adjacent, but this is not permitted by the tiling system: $\forall o_1, o_2 \in O : (o_1, o_2) \notin H \to O_\#^* o_1 o_2 O_\#^*$.

4. Two tiles are vertically adjacent, but this is not permitted by the tiling system: $\forall o_1, o_2 \in O : (o_1, o_2) \notin V \to O_\#^* o_1 O_\#^{2^n..2^n} o_2 O_\#^*$.

These regular expressions capture all strings that do not encode a tiling or violate the horizontal or vertical constraints of τ. Let $e_1, \dots, e_k$ be an enumeration of the above expressions. Let $e = e_1 \cdots e_k$. Because every e_i starts and ends with $O_\#^*$, $L(e) \subseteq L(e_i)$, for every i. We now define r_1 and r_2. Let $r_1 = \$e\$ \cdots \$e\$\#o_{bot}O_\#^* o_{top} O^*\#\$e\$ \cdots \$e\$$ with k-times $e\$$ at the beginning and k-times $e\$$ at the end of the string. Let $r_2 = \$O_{\$,\#}^*\$e_1\$e_2\$ \cdots \$e_k\$O_{\$,\#}^*\$$. Note that both regular expressions consist of a sequence of expressions of the form a, $(+a)^*$ and $(+a)^{i..j}$. So they are both in CHARE(S).

We must prove that $r_1 \subseteq r_2$ if and only if there does not exist a 2^n-corridor tiling for τ. Let $r_1 \subseteq r_2$. Then for every string $uwu' \in r_1$ where $u, u' \in L(\$e\$e\$ \cdots \$e\$)$ and $w \in \#o_{bot}O_\#^* o_{top} O^*\#$, $uwu' \in r_2$. However, as r_2 contains $k + 3$ times the symbol \$, and every string in $L(r_1)$ starts and ends with \$, there is an i such that $w \in L(e_i)$. So w does not encode a correct tiling, and thus there does not exist a 2^n-corridor tiling for τ. Conversely,

assume that there is a $uwu' \in L(r_1)$ where $u, u' \in L(\$e\$e\$ \cdots \$e\$)$ that is not in r_2. Then $w \notin \bigcup_{i=1}^{k} L(e_i)$ and, hence, encodes a correct tiling.

The only thing left to show is that this reduction can be done in logarithmic space. Constructing e is just a matter of constructing every e_k. In items 1 and 2 we have to output every symbol in O, this requires one pass through O, which requires one pointer that only uses logarithmic space. We also have to generate the numbers 2^{n-1} and 2^{n+1} in binary. In the proof of Lemma 6.4, we already saw that that can be done by using $\log n$ space. In items 3 and 4, we have to keep two pointers to O, which both require logarithmic space. Checking if a tuple of two tiles is in H or V is easy. The expressions itself contain again the enumeration of all symbols of O and the number 2^n, and thus can be generated in logarithmic space. Now we know that every e_i and e can be generated in logarithmic space, it is straightforward to generate r_1 and r_2 in logarithmic space. We do have to generate e several times since it is too big to store. □

Theorem 6.12. *The equivalence problem for CHARE(S) is in EXPSPACE.*

Proof. This immediately follows from the fact that the equivalence problem for RE$(+, \cdot, *, \#)$ is EXPSPACE-complete (Theorem 6.2). It, however, is possible that this problem is complete for a lower complexity class. □

Theorem 6.13. *The intersection problem for CHARE(S) is PSPACE-complete.*

Proof. We know that the intersection problem for $RE(+, \cdot, *, \#)$ is in PSPACE (Theorem 6.8). Since CHARE(S) is a subset of RE$(+, \cdot, *, \#)$, it follows that the intersection problem for CHARE(S) is in PSPACE.

We know that the intersection problem for CHARE($S \backslash \{\#\}$) is PSPACE-hard (Theorem 6.10). Since CHARE($S \backslash \{\#\}$) is a subset of CHARE(S), it follows that the intersection problem for CHARE(S) is PSPACE-hard. □

Theorem 6.14. *([MNS04])*

- *The inclusion problem is coNP-complete for CHARE($a, a?$) and CHARE(a, a^*).*
- *The equivalence problem is in PTIME for CHARE($a, a?$) and CHARE(a, a^*).*
- *The intersection problem is NP-complete for CHARE($a, a?$) and CHARE($a, a*$).*

Next, we show that the equivalence problem remains in PTIME for CHARE($a, a?, a\#$). In order to do this, we need a normal form for CHARE($a, a?$, $a\#$) expressions, the *sequence normal form.* Since the regular expressions in CHARE($a, a?, a\#$) cannot contain +-tokens, we only need to know the minimal and maximal number of occurrences of the base symbols in the

regular expression. Let $e[i, j]$ denote a sequence of at least i and at most j occurrences of the base symbol e.

In order to construct the sequence normal form of an expression, we write $e[1, 1]$ for e, $e[0, 1]$ for $e?$, and $e[n, m]$ for $e^{n..m}$. After this, we combine successive factors $e[i_1, j_1]$ and $e[i_2, j_2]$ into $e[i_1 + i_2, j_1 + j_2]$ whenever possible. For example, the sequence normal form of $a?a^{2..4}bb?a^{3..10}a$ is $a[2..5]b[1, 2]a[4, 11]$.

Finally we introduce some notions. If f is an expression $e[i, j]$, we write $e(f)$ for e, $l(f)$ for the lower bound i and $u(f)$ for the upper bound j. If $r = a_1[i_1, j_1] \cdots a_n[i_n, j_n]$ is an expression in sequence normal form, say $maxlength(r) = \sum_{k=1}^{n} j_k$. That is, the maximum length a string matched by r can have. The length of the expression $r = r_1 \cdots r_n$ in sequence normal form is denoted by $length(r)$ and is equal to n. Finally, we call a substring v of a string w a *block* of w when w is of the form $a_1^{k_1} \cdots a_n^{k_n}$, where for each $i = 1, \dots, n-1$ $a_i \neq a_{i+1}$ and v is of the form $a_i^{k_i}$ for some i.

Theorem 6.15. *The equivalence problem for CHARE$(a, a?, a\#)$ is in PTIME.*

Proof.

Lemma 6.16. *Two CHARE$(a, a?, a\#)$-expressions r_1 and r_2 are equivalent if and only if their sequence normal form is the same.*

Proof. In [MNS04] it is shown that two CHARE$(a, a?)$ expressions are equivalent if and only if their sequence normal form is the same[1]. We can write every CHARE$(a, a?, a\#)$-expression as an CHARE$(a, a?)$-expression by replacing every factor of the form $b^{i..j}$ by $b \cdots bb? \cdots b?$ with i times b and $j - i$ times $b?$. Let r'_1 and r'_2 be the CHARE$(a, a?)$ expressions by applying this to r_1 and r_2. We see that the normal forms of r_1 (resp. r_2) and r'_1 (resp r'_2) are equal. Since r'_1 and r'_2 are equivalent if and only if their sequence normal form is the same, the lemma follows. □

The algorithm to solve the equivalence problem is obvious. Given two regular expressions r_1 and r_2, generate their sequence normal form r'_1 and r'_2. If these are the same accept, else reject. By Lemma 6.16 this algorithm is correct.

It only remains to show that the sequence normal form of a CHARE$(a, a?, a\#)$-expression r can be generated in polynomial time. We first run through r one time and rewrite the factors of r into factors of the form $e[i, j]$. After this we run through the new string to combine successive factors. This

[1] In fact, it is shown that two CHARE$(a, a?, a*)$ expressions are equivalent if their strong sequence normal form is the same, where the strong sequence normal form is the sequence normal form modulo some rewrite rules. However, for CHARE$(a, a?)$ expressions, the strong and normal sequence normal form are always the same.

also requires only one pass through the string. Finally, we must compare the two generated normal forms, which can also be done in linear time. □

Theorem 6.17. *The intersection problem for CHARE($a, a?, a\#$) is NP-complete.*

Proof. Before we can prove that the intersection problem for CHARE($a, a?, a\#$) is in NP, we need a new way to represent a string.

Definition 6.18. *([MNS04]) A* compressed string *is a finite sequence of pairs (a, i), where $a \in \Sigma$ is a symbol and $i > 0$ is a natural number. The pair (a, i) stands for the string a^i. The size of a pair (a, i) is $\lceil \log i \rceil$, plus the size of a. The size of a compressed string $v = (a_1, i_1), \ldots, (a_n, i_n)$ is the sum of the sizes of $(a_1, i_1), \ldots, (a_n, i_n)$. The length of the compressed string is the number of pairs (a, i) it consists of and is denoted by $length(v)$. By* string(v)*, we denote the* decompressed *string corresponding to v, which is the string $a_1^{i_1} \ldots a_n^{i_n}$. Note that string(v) can have a size exponentially larger than v.*

Lemma 6.19. *If there is a string that is accepted by a number of CHARE(a, $a?, a\#$) expressions $r_1, \ldots, r_n$, we can rewrite it as a compressed string $v = (a_1, j_1), \ldots, (a_m, j_m)$, for which $m \leq min\{length(r_k) \mid 1 \leq k \leq n\}$ and $j_l \leq min\{maxlength(r_k) \mid 1 \leq k \leq n\}, 1 \leq l \leq m$.*

Proof. Suppose that there exists a string w, with m blocks which is accepted by every r_k, for $k = 1, \ldots, n$. We first rewrite it as a compressed string $v = (a_1, j_1), \ldots, (a_m, j_m)$. Since it is matched by every expressions r_k and since a factor of any r_i can never match more than one block of v, m can never have more blocks than the length of any expression r_k. Thus, $m \leq min\{length(r_k)\}$. Furthermore, since v is matched by every r_i, no j_l can be bigger than $maxlength(r_k)$, for any k, l. So we get $j_l \leq min\{maxlength(r_k) \mid 1 \leq k \leq n\}, 1 \leq j \leq m$. □

We can now prove that the intersection problem for CHARE($a, a?, a\#$) is in NP. We have to give a non-deterministic polynomial time algorithm which decides for a number of regular expressions $r_1, \ldots, r_n$ over $RE(a, a?, a\#)$ if the intersection of these regular expressions is non-empty. So our algorithm accepts when it finds a string that is accepted by all regular expressions. The algorithm begins by expressing all regular expressions in their sequence normal form. After this, we guess a compressed string and check if it is accepted by all regular expressions. By Lemma 6.19, we know that we only have to generate strings that satisfy the restrictions given in Lemma 6.19.

We start by guessing the length m of the compressed string v, an integer between 0 and $min\{length(r_i) \mid 1 \leq i \leq n\}$. Then, for each k, for which

$1 \leq k \leq m$, we guess a symbol $a_k \in \Sigma$ and an integer j_k, for which $0 < j_k \leq min\{maxlength(r_i) \mid 1 \leq i \leq n\}$.

We have to check if v is accepted by every expression $r_1, \ldots, r_n$. We do this by guessing an assignment of v for every regular expression. For an expression r_k in sequence normal form with length l, an *assignment* of v for r_k is a sequence of l integers $b_1 \cdots b_l$. This assignment must be interpreted as follows: The first b_1 symbols in v are matched by the first expression of r_k, the b_1+1th to b_1+b_2th symbols in v are matched by the second expression of r_k and so on. Note that no b_i can be bigger than $maxlength(r_i)$, since an expression r_k can never match more than $maxlength(r_i)$ symbols. So we guess an assignment $b_1 \cdots b_l$ for each r_k in which for each b_i, $0 \leq b_i \leq maxlength(r_k)$.

We can check if an assignment $b_1 \cdots b_l$ of v for $r_k = e_1 \cdots e_l$ is valid. This can be done by running simultaneously through these three strings. For each b_i, if not $l(e_i) \leq b_i \leq u(e_i)$ reject. If one of the first b_i symbols of v is not equal to $e(e_i)$, reject. Remove the first b_i symbols of v and move on to b_{i+1}. If we reach the end of the string and v is not empty, reject. If we have not rejected now, the assignment is valid and v is accepted by r_k.

If all assignments are valid, we have found a string which is accepted by all regular expressions and we accept.

If there exists a string which is accepted by all regular expressions, there will be a run of the algorithm in which that string is guessed and in which a valid assignment for every regular expression is guessed. On the contrary, if no such string exists, and thus the intersection of the regular expressions is empty, we won't be able to guess a string and a valid assignment for every regular expression.

This algorithm has to operate in polynomial time. Constructing the normal forms of the regular expressions can be done by running two times through the strings. Guessing the string v consists of guessing m (a polynomial) times a symbol and an integer between 0 and $min\{maxlength(r_k) \mid 1 \leq k \leq n\}$. Since the integer has a binary encoding, it always has a polynomial size. The total string v has a polynomial size. Finally, we must guess an assignment for every regular expression and check if it is valid for that regular expression. For any r_k, every assignment consists of m integers between 0 and $maxlength(r_k)$. Every assignment can thus be generated in polynomial time and has polynomial size. Checking if the assignment for any r_k is valid can be done by running simultaneously through the string v, the expression r_k and its assignment, which are all of polynomial size.

Since CHARE($a, a?$) is a strict subset of CHARE($a, a?, a\#$) and intersection for CHARE($a, a?$) is NP-hard (Theorem 6.14), intersection for CHARE($a, a?, a\#$) also is NP-hard. □

We see that the addition of the the numerical occurrence indicator to CHARE($a, a?$), did not affect the complexity of the equivalence and intersection problem. We conjecture that the same holds for the inclusion problem, but we haven't been able to prove it. Here, the results that immediately follow from the other theorems are presented.

Theorem 6.20. *The inclusion problem for CHARE($a, a?, a\#$) is coNP-hard, and in EXPSPACE.*

Proof. Since CHARE($a, a?$) is a strict subset of CHARE($a, a?, a\#$) and inclusion for CHARE($a, a?$) is coNP-hard (Theorem 6.14), inclusion for CHARE($a, a?, a\#$) also is coNP-hard.

Conversely, since CHARE($a, a?, a\#$) is a strict subset of $RE(+, \cdot, *, \#)$, and inclusion for $RE(+, \cdot, *, \#)$ is in EXPSPACE (Theorem 6.2), it follows that inclusion for CHARE($a, a?, a\#$) also is in EXPSPACE. □

Theorem 6.21. *The equivalence, inclusion and intersection problem for CHARE($a, a\#^{>0}$) are in PTIME.*

Proof. We show that the equivalence (Lemma 6.22), inclusion (Lemma 6.23) and intersection problem (Lemma 6.24) for CHARE($a, a\#^{>0}$) are in PTIME. □

Lemma 6.22. *The equivalence problem for CHARE($a, a\#^{>0}$) is in PTIME.*

Proof. We already know that the equivalence problem for CHARE($a, a?, a\#$) is in PTIME (Theorem 6.15). Since every CHARE($a, a\#^{>0}$)-expression also is a CHARE($a, a?, a\#$)-expression, the lemma immediately follows. □

Lemma 6.23. *The inclusion problem for CHARE($a, a\#^{>0}$) is in PTIME.*

Proof. Given two $RE(a, a\#^{>0})$-expressions r_1 and r_2, we start by creating their sequence normal forms r_1' and r_2'.

Let's take a string x_1 accepted by r_1'. We know that every factor in r_1' must match at least one symbol and that the base symbols of two consecutive factors in r_1' cannot be the same. Knowing this it is easy to see that the number of blocks in x_1, and thus in every string accepted by r_1, must be equal to $length(r_1')$. The same holds for r_2'. Following this we see that if $length(r_1') \neq length(r_2')$, $L(r_1)$ cannot be a subset of $L(r_2)$.

If the length of the two sequence normal forms is equal, we have to investigate the factors. Say $r_i' = e_{i,1}[l_{i,1}, u_{i,1}] \dots e_{i,n}[l_{i,n}, u_{i,n}]$, $i = 1, 2$. Every compressed string x accepted by r_1 and/or r_2 will be of the form $a_1^{k_1} \dots a_n^{k_n}$. Furthermore is every block of x matched by precisely one factor of r_1' and/or r_2'. More precisely, the jth block, $0 < j \leq n$, of x will be matched by the jth

factor of r'_1 and/or r'_2. It follows that if for some j, $0 < j \leq n$, $e_{1,j} \neq e_{2,j}$, $l_{1,j} > l_{2,j}$ or $u_{1,j} < u_{2,j}$ we can construct a string s with its jth block that is matched by r'_2 but not by r'_1. If $e_{1,j} = e_{2,j}$, $l_{1,j} < l_{2,j}$ and $u_{1,j} > u_{2,j}$ for every j, $0 < j \leq n$, we can never construct a block that is only matched by r'_2 and we can thus never construct a string that is accepted by r'_2 but not by r'_1.

This gives us the following algorithm for the inclusion problem:

1. Given two RE($a, a\#^{>0}$)-expressions r_1 and r_2 , construct their sequence normal forms r'_1 and r'_2.

2. If $length(r'_1) \neq length(r'_2)$, REJECT.

3. Say $r'_i = e_{i,1}[l_{i,1}, u_{i,1}] \dots e_{i,n}[l_{i,n}, u_{i,n}]$, $i = 1, 2$.

4. If $\exists j, 0 < j \leq n, e_{1,j} \neq l_{2,j}, l_{1,j} > l_{2,j}$, or $u_{1,j} < u_{2,j}$, REJECT. Else, ACCEPT.

This algorithm clearly operates in polynomial time. In step 1 we construct the sequence normal forms, this can be done in polynomial time. In step 4 we have to run one time through the sequence normal forms of polynomial size. This also imposes no problem.

□

Lemma 6.24. *The intersection problem for CHARE($a, a\#^{>0}$) is in PTIME.*

Proof. To test whether the intersection of a number of regular expressions $r_1, \dots, r_m$ is non-empty, we can follow the same logic as in Lemma 6.23. We also begin by creating their sequence normal forms $r'_1, \dots, r'_n$. If a string x is accepted by all regular expressions, the length of their sequence normal forms must all be equal to the number of blocks in x. Thus, if the length of all the sequence normal forms isn't the same, the intersection must be empty.

Let $r'_i = e_{i,1}[l_{i,1}, u_{i,1}] \dots e_{i,m}[l_{i,m}, u_{i,m}]$, $i = 1, \dots, n$. The intersection of all regular expression is non-empty if for every j, $0 < j \leq m$ we can construct a block that is matched by the jth factor of every r'_i. The string that is accepted by all regular expressions is the sequence of these blocks. If there is a j, $0 < j \leq m$, for which we cannot construct such a block it is not possible to construct a string that is accepted by all regular expressions. Given a j, we can construct such a block if $\max\{l_{i,j} \mid 0 < i \leq n\} \leq \min\{u_{i,j} \mid 0 < i \leq n\}$.

This gives us the following algorithm for the intersection problem:

1. Given n *RE*($a, a\#^{>0}$)-expressions $r_1, \dots, r_n$, construct their sequence normal forms $r'_n, \dots, r'_n$.

2. If $length(r'_k) \neq length(r'_l)$, for some k, l, $0 < k, l \leq n$, REJECT.

3. Say $r'_i = e_{i,1}[l_{i,1}, u_{i,1}] \dots e_{i,m}[l_{i,m}, u_{i,m}]$, $i = 1, \dots, n$.

4. If $\exists j, 0 < j \leq m : min\{u_{i,j} \mid 1 \leq i \leq n\} < max\{l_{i,j} \mid 1 \leq i \leq n\}$, REJECT. Else, ACCEPT.

This algorithm works in polynomial time. We only have to construct the sequence normal forms of the given regular expressions and read them once. This can all be done in polynomial time. □

Chapter 7

Complexity of unranked tree automata

In this chapter, we investigate the equivalence, inclusion and intersection problem for unranked tree automata. The complexity of these problems depends on the class of regular expressions allowed in the UTA. The results of this chapter are summarized in Table 7.1.

	equivalence	inclusion	intersection
UTA(RE($+, \cdot, *$))	EXPTIME [Sei90] (7.1)	EXPTIME (7.1)	EXPTIME [Sei94] (7.9)
UTA(RE($+, \cdot, *, \#$))	EXPSPACE (7.1)	EXPSPACE (7.1)	EXPTIME (7.9)
UTA(RE($+, \cdot, *, \&$))	EXPSPACE (7.1)	EXPSPACE (7.1)	EXPTIME (7.9)
UTA(RE($+, \cdot, *, \#, \&$))	EXPSPACE (7.1)	EXPSPACE (7.1)	EXPTIME (7.9)

Table 7.1: Complexity classes for problems with unranked tree automata. All complexities are completeness results. The complexity of the equivalence and intersection problem for UTA(RE($+, \cdot, *$)) are due to Seidl. The theorem numbers are given in brackets.

We see that the equivalence, inclusion, and intersection problem for UTA(RE($+, \cdot, *$)) are EXPTIME-complete. When we add the numerical occurrence, shuffle, or both operators, the intersection problem remains EXPTIME-complete. The corresponding problems for regular expressions where all PSPACE-complete. So, in comparison to regular expressions, we go from space to time complexity, but go one exponential higher.

The complexity of the equivalence and inclusion problem for regular expression with the numerical occurrence or shuffle operator is EXPSPACE. Therefore, we expected the corresponding problems for unranked tree automata to be 2EXPTIME-complete. However, it turns out that the complex-

ity of the equivalence and inclusion problem for unranked tree automata extended with the numerical occurrence or shuffle operator also is EXPSPACE. Even when we add both operators, we remain in EXPSPACE.

Note that EXPTIME and EXPSPACE are closed under complement and therefore the same completeness results hold for the complement of the equivalence, inclusion and intersection problem.

7.1 Equivalence and inclusion

Theorem 7.1.

- *The equivalence and inclusion problem for UTA(RE$(+, \cdot, *)$) are EXPTIME-complete.*
- *The equivalence and inclusion problem for UTA(RE$(+, \cdot, *, \#, \&)$) are EXPSPACE-complete. These problems remain EXPSPACE-hard for UTA(RE$(+, \cdot, *, \#)$) and UTA(RE$(+, \cdot, *, \&)$).*

Proof. We prove this theorem in a series of lemmas. We first show that the equivalence and inclusion problem for UTA(RE$(+, \cdot, *)$) are in EXPTIME (Lemma 7.2). Seidl has already shown this for the equivalence problem ([Sei90]), but we use another technique which can also be used for UTA(RE$(+, \cdot, *, \#, \&)$)s. Then, we show how we can adjust the algorithm of Lemma 7.2 to prove that the equivalence and inclusion problem for UTA(RE($+, \cdot, *, \#, \&$)) are in EXPSPACE (Lemma 7.4). In Lemma 7.6, we show that the inclusion and equivalence problem for UTA(RE$(+, \cdot, *)$) are EXPTIME-hard. Finally, we prove that the equivalence and inclusion problem for UTA(RE$(+, \cdot, *, \#)$) and UTA(RE$(+, \cdot, *, \&)$) are EXPSPACE-hard (Lemma 7.8). Together, these lemmas prove the theorem.

Lemma 7.2. *The equivalence and inclusion problem for UTA$(+, \cdot, *)$ are in EXPTIME.*

Proof. We give an EXPTIME algorithm which decides the equivalence problem for UTA$(+, \cdot, *)$. The algorithm for the inclusion problem is analogue. On input two UTA$(+, \cdot, *)$s $A_1 = (Q_1, \Sigma, \delta_1, F_1)$, $A_2 = (Q_2, \Sigma, \delta_2, F_2)$, we have two decide if $L(A_1) = L(A_2)$.

To this end, we construct a set E of tuples (C_1, C_2), where $C_1 \subseteq Q_1$ and $C_2 \subseteq Q_2$. Here, E contains a tuple (C_1, C_2) if and only if there exists a tree t such that

1. for every $q \in C_1$, there exists a run λ of A_1 on t such that $\lambda(\varepsilon) = q$. That is, q is the state at the root for a run of A_1 on t;

2. for every $q \in Q_1$, $q \notin C_1$, there does not exist a run λ of A_1 on t such that $\lambda(\varepsilon) = q$;

3. for every $q \in C_2$, there exists a run λ of A_2 on t such that $\lambda(\varepsilon) = q$; and

4. for every $q \in Q_2$, $q \notin C_2$, there does not exist a run λ of A_2 on t such that $\lambda(\varepsilon) = q$;

Intuitively, if a set $(C_1, C_2) \in E$, this means that there exists a tree t such that the states in C_1 (resp. C_2) can be the only states at the root node for any run of A_1 (resp. A_2) on t. We will also say that t *generates* the set (C_1, C_2). Note that these sets C_1 and C_2 contain all states that can be reached at the root node, and not just the accepting state. However, A_1 (resp. A_2) accepts t if and only if $C_1 \cap F_1$ (resp. $C_2 \cap F_2$) is not empty.

Suppose that we have computed E, and E contains the tuples (C_1, C_2) for all possible trees. We can now search E for a tuple (C_1, C_2) such that $C_1 \cap F_1 \neq \emptyset$, and $C_2 \cap F_2 = \emptyset$. If there exists such a tuple, we know that there exists a tree t such that $t \in L(A_1)$ and $t \notin L(A_2)$, which proves that $L(A_1) \neq L(A_2)$. We also do the opposite. We search for a tuple (C_1, C_2) such that $C_1 \cap F_1 = \emptyset$, and $C_2 \cap F_2 \neq \emptyset$, which also shows that $L(A_1) \neq L(A_2)$. Therefore, when we have computed E, we search all tuples for one of the two previous properties to show that $L(A_1) \neq L(A_2)$. If there does not exist a tuple that has one of the two previous properties, we know that every possible tree is accepted by A_1 and A_2, or rejected by A_1 and A_2.

We construct the set E recursively. We first compute every possible element of E for every tree of depth 1 over Σ. Then, given the set E which contains all tuples for trees with depth $\leq m$, we add all tuples for trees with depth $\leq m + 1$. We repeat this last step until no tuples are added to E anymore. At that point, we know that E will not grow anymore, and contains all possible tuples we searched for.

We describe the different steps in more depth. We first have to generate all tuples for trees of depth 1. For every $a \in \Sigma$, we add a tuple (C_1, C_2) to E, where for $i = 1, 2$, $C_i = \{q \mid \varepsilon \in \delta(q, a)\}$. We can decide whether $\varepsilon \in \delta(q, a)$, by constructing an NFA A such that $L(A) = L(\delta(q, a))$. For A, we check if we can reach an accept state of A by using only ε-transitions. This can be done in PTIME.

For the induction step, we have the set E, which already contains all tuples (C_1, C_2), such that there exists a tree with depth $\leq m$ which generates the tuple (C_1, C_2). For any tuple (C_1, C_2), not yet in E, we have to decide whether there exists a tree with depth $\leq m + 1$, which generates (C_1, C_2). In

Proposition 4.14, we show that this can be done in EXPTIME. Finally, we add all tuples that are generated by a tree with depth $\leq m+1$ to E.

We show that this algorithm is in EXPTIME. The basis step only requires a polynomial number of steps. In every induction step, we have to decide for every tuple which is not yet in E, if we have to add it. In Proposition 7.3, we show that deciding this for one tuple can be done in exponential time. Furthermore, there are only an exponential number of different values for C_1 and C_2, since $C_1 \subseteq Q_1$ and $C_2 \subseteq Q_2$. It follows that the number of tuples (C_1, C_2) also is an exponential since we have an exponential number of combinations times an exponential number of combinations, which remains exponential. Therefore, in every induction step, we maximally have to execute an exponential time algorithm, exponentially many times. This again is an exponential.

The question is how many times we have to execute the induction step. We know that every induction step has to add at least one tuple. If no tuple is added in the induction step, we stop the algorithm. However, we just showed that there only an exponential number of different tuples. Therefore, we maximally have to execute the induction step an exponential number of times. All together, this again is an exponential. Finally, we have to check a few properties of an exponential number of tuples. This can also be done in exponential time.

The following lemma concludes the proof of this theorem.

Proposition 7.3. *We are given the set E, which contains every tuple (C_1', C_2'), such that there exists a tree with depth $\leq m$ which generates the tuple (C_1', C_2') for UTAs A_1, and A_2. Then, we can decide for a tuple (C_1, C_2) whether it is generated by a tree with depth $\leq m+1$. Furthermore, this can be done in time exponential in the size of A_1 and A_2.*

Proof idea We construct a non-deterministic polynomial space algorithm that solves the problem. Since NPSPACE $\subseteq$ EXPTIME, the proposition follows.

We try to guess a tree t that generates (C_1, C_2). This tree has a symbol $a \in \Sigma$ as label of the root node, and this root node has a number of children, which form trees with depth $\leq m$, say $t_1, \ldots, t_n$. The tree t is then of the form $a(t_1 \cdots t_n)$. Every subtree t_i, $i = 1, \ldots, n$, generates a tuple (C_1^i, C_2^i) of states, which A_1 and A_2 can be in at the root of t_i. For $i = 1, 2$, we see that A_i can be in a state q at the root node of t, if there exists a $w \in C_i^1 \cdots C_i^n$, such that $w \in \delta_i(q, a)$. Here, $C_i^1 \cdots C_i^n$ is a regular expression, where every C_i^j stands for the disjunction of all its elements.

So, to check if (C_1, C_2) is generated by t, we have to check if for every $q_1 \in C_1$ (resp. $q_2 \in C_2$), there exists a string $w_1 \in C_1^1 \cdots C_1^n$ (resp. $w_1 \in$

$C_2^1 \cdots C_2^n$) such that $w_1 \in \delta_1(q_1, a)$ (resp. $w_2 \in \delta_2(q_2, a)$). And, for every $q_1 \in Q_1$, $q_1 \notin C_1$ (resp. $q_2 \in Q_2$, $q_2 \notin C_2$), there may not exist such a string w_1 (resp. w_2). To check if such strings do or do not exist, we use NFAs. We construct NFAs for the transition functions $\delta_1(q, a)$, $\delta_2(q, a)$. Note that the alphabet of these NFAs is the set of states of the UTAs A_1 and A_2. For the states q which are in C_1 or C_2, we only have to show that there exists a string w which is accepted by its corresponding NFA. We use non-determinism to guess the string w and guess the accepting run of that automaton on w. Therefore, we only maintain one possible state that automaton can be in. For the states q which are not in C_1 or C_2, we have to be sure that there does not exist a w which is accepted by its corresponding automaton. Therefore, we can not just guess one w, and one run of that automaton on w. Instead, we have to maintain a set of all possible states that automaton can be in after reading any possible w. If this set does not contain an accept state, this means that the automaton does not accept any possible w.

Proof. The following non-deterministic polynomial space algorithm decides whether there exists a tree t with depth $\leq m + 1$, which generates (C_1, C_2) for A_1 and A_2.

1. Guess an $a \in \Sigma$, the label of the root node of t.
2. Construct the various NFAs based on a and the transition functions of A_1 and A_2:
   (a) For every $q \in C_1$, construct an NFA B_1 such that $L(\delta_1(q, a)) = L(B_1)$. Denote these NFAs by $B_1^1, \ldots, B_1^{k_1}$, where $k_1 = |C_1|$, and for $i = 1, \ldots, k_1$, $B_1^i = (Q_1^i, \Sigma_1^i, q_1^i, f_1^i, \delta_1^i)$.
   (b) For every $q \in Q_1$, $q \notin C_1$, construct an NFA $\overline{B}_1$ such that $L(\delta_1(q, a)) = L(B_1)$. Denote these NFAs by $\overline{B}_1^1, \ldots, \overline{B}_1^{l_1}$, where $l_1 = |Q_1 \backslash C_1|$, and for $i = 1, \ldots, l_1$, $\overline{B}_1^i = (\overline{Q}_1^i, \overline{\Sigma}_1^i, \overline{q}_1^i, \overline{f}_1^i, \overline{\delta}_1^i)$.
   (c) For every $q \in C_2$, construct an NFA B_2 such that $L(\delta_2(q, a)) = L(B_2)$. Denote these NFAs by $B_2^1, \ldots, B_2^{k_2}$, where $k_2 = |C_2|$, and for $i = 1, \ldots, k_2$, $B_2^i = (Q_2^i, \Sigma_2^i, q_2^i, f_2^i, \delta_2^i)$.
   (d) For every $q \in Q_2$, $q \notin C_2$, construct an NFA $\overline{B}_2$ such that $L(\delta_2(q, a)) = L(B_2)$. Denote these NFAs by $\overline{B}_2^1, \ldots, \overline{B}_2^{l_2}$, where $l_2 = |Q_2 \backslash C_2|$, and for $i = 1, \ldots, l_2$, $\overline{B}_2^i = (\overline{Q}_2^i, \overline{\Sigma}_2^i, \overline{q}_2^i, \overline{f}_2^i, \overline{\delta}_2^i)$.
3. For $i = 1, 2$, $j = 1, \ldots, k_i$, let $s_i^j = q_i^j$.
4. For $i = 1, 2$, $j = 1, \ldots, l_i$, let $\overline{S}_i^j = \{\overline{q}_i^j\}$.

5. For $i = 1, 2$, $j = 1, \dots, k_i$, guess, if possible, whether to follow an epsilon transition $\delta_i^j(s_i^j, \varepsilon) = s'^j_i$, and set $s_i^j = s'^j_i$. Guess whether to repeat step 5 (GOTO 5) or continue with the algorithm (GOTO 6).

6. For $i = 1, 2$, $j = 1, \dots, l_i$, compute $\overline{S'}^j_i = \{q \mid \overline{\delta}_i^j(q_s, \varepsilon) = q, q_s \in \overline{S}_i^j\}$, and let $\overline{S}_i^j = \overline{S}_i^j \cup \overline{S'}^j_i$. Repeat step 6 until no $\overline{S}_i^j$ changes anymore.

7. If, for every $i = 1, 2$, $j = 1, \dots, k_i$, $s_i^j = f_i^j$, and for every $i = 1, 2$, $j = 1, \dots, l_i$, $\overline{f}_i^j \notin \overline{S}_i^j$, ACCEPT.

8. Guess a tuple $(C'_1, C'_2) \in E$. This tuple represents the next child or subtree of t.

9. For $i = 1, 2$, $j = 1, \dots, k_i$, guess a $p \in C'_i$, guess, if possible, a transition $\delta_i^j(s_i^j, p) = s'^j_i$, and set $s_i^j = s'^j_i$. If no such transition exists, REJECT.

10. For $i = 1, 2$, $j = 1, \dots, l_i$, compute $\overline{S'}^j_i = \{q \mid \overline{\delta}_i^j(q_s, p) = q, q_s \in \overline{S}_i^j, p \in C'_i\}$, and let $\overline{S}_i^j = \overline{S'}^j_i$.

11. GOTO 5.

We describe the algorithm step by step. We begin by guessing the label of the root of t (step 1). Then, we create NFAs based on a and the transition functions of A_1 and A_2 (step 2). The B_i^j automata represent those states that are in the C_i. The $\overline{B}_i^j$ automata represent those states that are not in the C_i. For the B_i^j we only have to guess one string which is accepted. Therefore, we only maintain one state s_i^j and guess the accepting run of B_i^j on one guessed string. For the $\overline{B}_i^j$, we have to check if none of the possible strings is accepted, therefore we maintain sets $\overline{S}_i^j$ of all possible states the $\overline{B}_i^j$ can be in.

In steps 3 and 4, we initialize the s_i^j and $\overline{S}_i^j$ with their start states. Then, we allow every B_i^j to follow a random number of ε-transitions (step 5). In step 6, we update the $\overline{S}_i^j$ such that the $\overline{S}_i^j$ contain all states that can be reached by only following ε-transitions. Next, we check if every B_i^j accepts, and no $\overline{B}_i^j$ accepts (step 7). If this is the case, we have found a tree that generates (C_1, C_2) and can accept. If we can not accept, we guess the next child of the root node represented by the tuple (C'_1, C'_2). For the B_i^j, we guess a state from C'_i, and follow a transition using that state (step 9). For the $\overline{B}_i^j$, we follow every possible transition using the states in C'_i, and update the $\overline{S}_i^j$ accordingly (step 10). Then, we return to step 5 to follow ε-transitions, and to check if we can accept or guess a next child.

If this algorithm accepts, we have guessed a tree t with as root label a, and as children $t_1, \ldots, t_n$, where for $i = 1, \ldots, n$, (C_1^i, C_2^i) is generated by t_i. For $i = 1, 2$, we have shown that for every $q \in C_i$, there exists a $w \in C_i^1 \cdots C_i^n$ such that $w \in \delta_i(q, a)$. It follows that there exists a run λ of A_i on t, for which $\lambda(\varepsilon) = q$. We have also shown that for every $q \in Q_i$, $q \notin C_i$, there does not exist a $w \in C_i^1 \cdots C_i^n$ such that $w \in \delta_i(q, a)$. It follows that there dos not exist a run λ of A_i on t, for which $\lambda(\varepsilon) = q$. Furthermore, since every guessed tuple (C_1^i, C_2^i) represents a tree t_i with depth $\leq m$, we know that t has depth $\leq m + 1$.

Conversely, suppose that there exists a tree t with depth $\leq m + 1$, such that t generates (C_1, C_2). Suppose that t is of the form $a(t_1 \cdots t_n)$, where every t_i generates the tuple (C_1^i, C_2^i). Since t has depth $\leq m + 1$, we know that every t_i must have depth $\leq m$. Furthermore, we know that E contains all tuples generated by trees with depth $\leq m$. Therefore, there exists a run of our algorithm which (i) guesses the rootlabel a of t, (ii) guesses the tuples (C_1^i, C_2^i) representing the t_i, and (iii) guesses the accepting runs of the B_i^j. Furthermore, since we have constructed the $\overline{B}_i^j$ for the states not in C_1 and C_2, and t generates exactly (C_1, C_2), none of the $\overline{B}_i^j$ can accept after guessing the nth child. This run of our algorithm accepts.

Finally, we have to show that this algorithm operates in polynomial space. We create a polynomial number of NFAs $(|Q_1| + |Q_2|)$, which are all of polynomial size (Theorem 4.6). For these NFAs we maintain one state s_i^j, or a set of states $\overline{S}_i^j$. These variables or states are all polynomial in the size of the NFAs, which are themselves polynomial in the size of A_1 and A_2. The variables ${s'}_i^j$, and $\overline{S'}_i^j$ require an equal amount of space as s_i^j, and $\overline{S}_i^j$, respectively. The computations performed in the different steps of the algorithm can all be done in polynomial space.

□

We have shown that the equivalence problem for UTA$(+, \cdot, *)$ is in EXPTIME. The algorithm for the complement of the inclusion problem is analogue. We generate the same set E. The only difference is that if we want to check if $L(A_1) \not\subseteq L(A_2)$, we have to check if there is a tuple $(C_1, C_2) \in E$, such that $C_1 \cap F_1 \neq \emptyset$ and $C_1 \cap F_1 = \emptyset$. If that is the case, then there exists a tree t such that $t \in L(A_1)$ and $t \notin L(A_2)$. If there does not exist such a tuple, we know that $L(A_1) \subseteq L(A_2)$. Since EXPTIME is closed under complement, it follows that the inclusion problem for UTA$(+, \cdot, *)$ is in EXPTIME.

□

Lemma 7.4. *The equivalence and inclusion problem for UTA$(RE(+, \cdot, *, \#, \&))$ are in EXPSPACE.*

Proof. We can use the same algorithm as we did for the algorithms for the equivalence and inclusion problem for UTA(RE($+, \cdot, *, \#, \&$)), which are in EXPTIME (Lemma 7.2). The only difference is that the UTAs are now allowed to use the numerical occurrence and shuffle operator in their transition functions. Therefore, we have to use ENFAs instead of NFAs, which takes our algorithm to EXPSPACE.

Since these algorithms are so similar, we only describe the differences between the two algorithms. In the basis step, we have to check whether $\varepsilon \in r$, where r is a RE($+, \cdot, *, \#, \&$)-expression. We have already shown that we can do this in space polynomial of the size of r (Lemma 4.14). In the induction step, we have to create a polynomial number of ENFAs B_i^j and $\overline{B}_i^j$ equivalent to RE($+, \cdot, *, \#, \&$)-expressions instead of NFAs equivalent to RE($+, \cdot, *$)-expressions. These ENFAs can be constructed in time polynomial in the size of the regular expression and therefore have a polynomial size (Lemma 4.13). The variables s_i^j will now contain a configuration of the B_i^j, instead of a state, and the variables $\overline{S}_i^j$ contain a set of configurations of the $\overline{B}_i^j$. One configuration of any of the B_i^js or $\overline{B}_i^j$s, has a size polynomial in the size of the B_i^js or $\overline{B}_i^j$s (Lemma 4.13). Therefore, the s_i^j can be stored in polynomial size, and the $\overline{S}_i^j$ can be stored in exponential size. The latter follows from the fact that there can only exist an exponential number different polynomial size configurations. Since an exponential number of polynomially sized configurations can be stored in exponential space, the $\overline{S}_i^j$ can be stored in exponential space. The computation of these variables s_i^j and $\overline{S}_i^j$ in the different steps of the algorithm of the induction step can be done in a similar way as we did for NFAs. We can also follow ε-transitions and normal transitions as we do for NFAs. These computations can of course be done in exponential space.

If we combine this information, we see that it requires exponential space to check for one tuple if it is generated by a tree with depth $\leq m + 1$ in the induction step. In one induction step, we maximally have to do this an exponential number of times. Since using exponential space an exponential number of times also requires exponential space, every induction step can be done in exponential space. We have also argued that we maximally have to do an exponential number of induction steps. Therefore, the total algorithm can be done in exponential space. □

The following lemma is used in the proof of Lemma 7.6.

Lemma 7.5. *Given a NFA A, we can construct an UTA(RE($+, \cdot, *$)) B, which accepts a tree t if and only if t contains a path which is accepted by A.*

Proof. Let $A = (Q, \Sigma, q_0, q_f, \delta)$ be a NFA. We construct $B = (Q', \Sigma', \delta', F')$ as follows:

- $Q' = Q \cup \{q_a\}$;
- $\Sigma' = \Sigma$;
- For any $q \in Q$, $\sigma \in \Sigma$, say $\delta(q, \sigma) = \{q_1, \dots, q_n\}$. If $q_f \notin \{q_1, \dots, q_n\}$, then $\delta'(q, \sigma) = q_a^*(q_1 + \cdots + q_n)q_a^*$. If $q_f \in \{q_1, \dots, q_n\}$, then $\delta'(q, \sigma) = q_a^*(q_1 + \cdots + q_n)q_a^* + \varepsilon$; and
- For any $\sigma \in \Sigma$, $\delta'(q_a, \sigma) = q_a^*$; and
- $F' = \{q_0\}$.

In this construction, B will guess a path in the tree, and accepts at the end of that path if A accepts that path. All other paths are always accepted. So, if a tree t contains a path that is accepted by A, that path will be guessed by a run of B, and t is accepted. If a tree is accepted by B, it must have a path that is accepted by A. □

Lemma 7.6. *The equivalence and inclusion problem for UTA($RE(+, \cdot, *)$) are EXPTIME-hard.*

Proof. We first prove that the equivalence problem for UTA($RE(+, \cdot, *)$) is EXPTIME-hard. The proof for the inclusion problem is analogue. We do a reduction from the 2-player corridor tiling problem, which is EXPSPACE-complete (Theorem 3.5). On input an instance $\tau = \langle O, \overline{b}, \overline{t}, n \rangle$, we generate UTA($RE(+, \cdot, *)$)s A_1 and A_2, such that $L(A_1) = L(A_2)$ if and only if CONSTRUCTOR has a winning strategy for the corridor tiling game on τ.

In this proof, we use a strategy tree for a corridor tiling game, as defined by Martens ([Mar06], Theorem 9.9). Every path in such a tree encodes a tiling. The label of the root is #, and every other node is labeled with a tile. Since CONSTRUCTOR and SPOILER place their tiles in turn, the tiles of CONSTRUCTOR are always located at even depth, and those of SPOILER at odd depth. The SPOILER tiles have exactly one child, which represents the choice of the CONSTRUCTOR at that point. The CONSTRUCTOR tiles have all possible tiles as children, representing all possible choices of SPOILER. Each of these tiles which is not valid, because it does not satisfy the horizontal or vertical constraints, has *error* as the label of its child. This shows that this is not a valid path. We can encode a winning strategy of CONSTRUCTOR in such a strategy tree. So, there exists a strategy tree that encodes a winning strategy of CONSTRUCTOR if and only if there exists a winning strategy for CONSTRUCTOR.

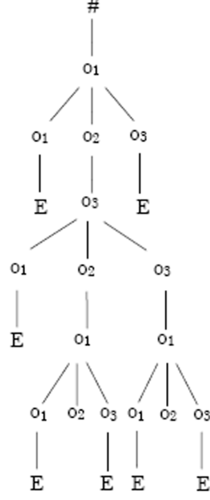

Figure 7.1: A strategy tree for the winning strategy of CONSTRUCTOR in the 2-player corridor tiling problem of Example 7.7.

Example 7.7. *Take the instance* $\tau = \langle O, \overline{b}, \overline{t}, n \rangle$*, where*

- $O = \{o_1, o_2, o_3\}$*;*
- $\overline{b} = \overline{t} = (o_1, o_2)$*;*
- $n = 2$*;*

where the following horizontal and vertical constraints hold

- $H = \{(o_1, o_2), (o_3, o_2), (o_3, o_3)\}$*; and*
- $V = \{(o_1, o_3), (o_2, o_2), (o_2, o_3), (o_3, o_1), (o_3, o_2)$*.*

For the 2-player corridor tiling problem on τ*, CONSTRUCTOR has a winning strategy. The strategy tree that encodes this winning strategy is shown in Figure 7.1. Depending on the choices of SPOILER, this strategy can lead to two valid corridor tilings. These are shown in Figure 7.2.* □

We define A_1 as the UTA which accepts all possible trees over the alphabet $\Sigma = O \cup \{\#, error\}$, and A_2 will be the UTA that accepts all possible trees over Σ except those that encode a strategy tree for a winning strategy of CONSTRUCTOR. If no such tree exists, and thus the 2-player corridor tiling problem rejects, A_2 is equal to A_1. Conversely, if the 2-player corridor tiling problem accepts, there exists such a strategy tree and A_2 is not equal to A_1.

We begin by describing $A_1 = (Q_1, \Sigma, \delta_1, F_1)$:

o_1	o_2		o_1	o_2
o_3	o_2		o_3	o_3
o_1	o_2		o_1	o_2

Figure 7.2: The two possible valid tilings that can be constructed from the strategy tree of Figure 7.1.

- $Q_1 = F_1 = \{q\}$;
- $\Sigma = O \cup \{\#, error\}$; and
- For any $\sigma \in \Sigma$: $\delta_1(q, \sigma) = q^*$.

We describe A_2 using a number of UTAs and NFAs which express certain errors in a strategy tree. We use these NFAs to check if a path of the tree, rather than the whole tree, contains an error. In Lemma 7.5, we have seen how we can construct an UTA, which accepts a tree if it contains a path which is accepted by a NFA. Finally, we take the union of these UTAs, which will be A_2. We now describe all possible errors in a tree that prevent it to be a correct strategy tree. Suppose that the set of tiles $O = \{o_1, \ldots, o_n\}$.

1. The tree t does not have $\#$ as root symbol, $\#$ occurs at any other position in t, $error$ occurs at a non-leaf node of t, t does not have exactly one choice for a tile of CONSTRUCTOR, or t does not have all possible tiles as choices for SPOILER. The following UTA, (Q, Σ, δ, F), checks these conditions:

   - $Q = \{q_v, q_a, q^c, q_1^s, \ldots, q_n^s\}$;
   - $F = \{q_v\}$;
   - $\delta(q_v, \#) = q^c$, and for any $\sigma \in \Sigma$, $\sigma \neq \#$, $\delta(q_v, \sigma) = q_a^*$;
   - For any $\sigma \in \Sigma$, $\delta(q_a, \sigma) = q_a^*$;
   - $\delta(q^c, \#) = q_a^*$, $\delta(q^c, error) = q_a^+$, and for every $\sigma \in \Sigma$, $\sigma \neq \#$, $\sigma \neq error$, $\delta(q^c, \sigma) = q_a^{1..n-1} + q_a^{n+1..n+1} q_a^* + q_1^s q_a^{n-1..n-1} + q_a q_2^s q_a^{n-2..n-2} + \cdots + q_a^{n-2..n-2} q_{n-1}^s q_a + q_a^{n-1..n-1} q_n^s$; and
   - For any $i = 1, \ldots, n$, $\delta(q_i^s, \#) = \delta(q_i^s, error) = q_a^*$, $\delta(q_i^s, o_i) = q_c + q_a q_a^+$. For any $o_i \in O$, $q_j^s \in Q$, $i \neq j$, $\delta(q_j^s, o_i) = q_a^*$.

   The state q_v always is the state at the root node, the nodes which contain a tile which is placed by CONSTRUCTOR are in state q_c, and the nodes which contain a tile that is placed by CONSTRUCTOR are in one of the states $q_1, \ldots, q_n$, depending on which tile, $o_1, \ldots, o_n$,

must be placed at that position. By maintaining this information in our states, we can search for misplaced labels and, if we find such an error, we immediately go into state q_a. The state q_a is used to denote that we have found an error and can accept. We explain the transition function $\delta(q^c, \sigma) = q_a^{1..n-1} + q_a^{n+1..n+1} q_a^* + q_1^s q_a^{n-1..n-1} + q_a q_2^s q_a^{n-2..n-2} + \cdots + q_a^{n-2..n-2} q_{n-1}^s q_a + q_a^{n-1..n-1} q_n^s$ a bit more in depth. A node which contains a tile that is placed by CONSTRUCTOR, must have exactly n child nodes which contain $o_1, \cdots, o_n$, in that order. The two first factors of this transition function make sure that if it has more or less than n children, we immediately accept. If there are exactly n children, we guess for which child node we will test if it contains a wrong tile. This is done by the last n factors of the transition function.

Note that our regular expression can only use the operators $\cdot$, $+$, and $*$. The numerical occurrence indicators used in these regular expressions are abbreviations, rather than operators.

2. The tree contains a path that does not end with *error*, and whose length is not equal to $1 + m \cdot n$, for any $m \in \mathbb{N}$. We construct a NFA, $(Q, \Sigma, q_0, q_f, \delta)$, which accepts such a path.

   - $Q = \{q_v, q_a, q_1, \ldots, q_n\}$;
   - $q_0 = q_v$;
   - $q_f = q_a$;
   - $\delta(q_v, \#) = q_1$;
   - For $i = 1, \ldots, i-1$, $\sigma \in \Sigma$, $\sigma \neq error$, $\delta(q_i, \sigma) = \{q_{i+1}, q_a\}$; and
   - For $\sigma \in \Sigma$, $\delta(q_n, \sigma) = \{q_1\}$.

   In this automaton, q_v is the start state and q_a is the accept state. The states q_1 to q_n are used to keep track of the position in the string modulo n. If we are at the last position of the string, we don't read 'error', and we are not in state q_n, we can accept.

3. The tree contains a path which encodes a tiling for which the bottom row is not equal to $\bar{b}$. A path encodes a tiling if it does not end with *error*. We construct a NFA, $(Q, \Sigma, q_0, q_f, \delta)$, which accepts such a path.

   - $Q = \{q_v, q_a, q_1, \ldots, q_n\}$;
   - $q_0 = q_v$;
   - $q_f = q_a$;

- $\delta(q_v, \#) = \{q_1\}$;
- For $i = 1, \ldots, n-1$, $\delta(q_i, \overline{b}(i)) = \{q_{i+1}\}$;
- For $i = 1, \ldots, n$, $\sigma \in \Sigma$, $\sigma \neq \overline{b}(i)$, $\delta(q_i, \sigma) = \{q_a\}$; and
- For $\sigma \in \Sigma$, $\sigma \neq error$, $\delta(q_a, \sigma) = q_a$.

The state q_v is again the start state. The states q_1 to q_n check if the first n tiles are at any position different from $\overline{b}$. If that is the case, we enter state q_a and remain in state q_a until the end of the string. If the last symbol of the string is not equal to 'error', we accept.

4. The tree contains a path which encodes a tiling for which the top row is not equal to $\overline{t}$. A path encodes a tiling if it does not end with *error*. We construct a NFA, $(Q, \Sigma, q_0, q_f, \delta)$, which accepts such a path.

- $Q = \{q_v, q_1^a, \ldots, q_{n+1}^a, q^r, q_1^r, \ldots, q_n^r\}$;
- $q_0 = q_v$;
- $q_f = q_{n+1}^a$;
- $\delta(q_v, \#) = q^r$;
- For any $\sigma \in \Sigma$, $\delta(q^r, \sigma) = \{q^r, q_1^r\}$;
- For $i = 1, \ldots, n-1$, $\delta(q_i^r, \overline{t}(i)) = \{q_{i+1}^r\}$;
- For $i = 1, \ldots, n$, $\sigma \in \Sigma$, $\sigma \neq \overline{t}(i)$, $\delta(q_i^r, \sigma) = \{q_{i+1}^a\}$; and
- For $i = 1, \ldots, n$, $\sigma \in \Sigma$, $\sigma \neq error$, $\delta(q_i^a, \sigma) = q_{i+1}^a$.

This automaton is almost the same as the previous one, but its a bit more complicated because we have to check the last n tiles instead of the first n tiles. The state q_v is again the start state, the states with a r in superscript denote that we are going to reject the string, those marked with a a denote that we are going to accept the string. The states with a subscript ranging from one to $n+1$ are used to control if the top row is equal to $\overline{t}$ and the subscripts denote the position in this top row.

We always begin in q_v, which immediately passes the control to state q^r. We stay in q^r until we guess that we have reached the first tile of the top row and enter q_1^r. From here on, the states q_1^r to q_n^r, and q_1^a to q_{n+1}^a are used to check if the top row is different from $\overline{t}$. If it is different, at any position, we arrive at the state q_{n+1}^a, and accept.

5. The tree contains a path in which CONSTRUCTOR violates a horizontal constraint.

From here on, we will only describe the automata informally. The formal constructions of the automata are straightforward.

The NFA reads the string, and non-deterministically chooses a position which contains a tile placed by SPOILER to check if the horizontal constraints between this tile and the next tile are respected. If they are not respected, we accept. The only place where we do not have to check the horizontal constraints between two successive tiles is between the last tile of one row, and the first tile of the next row. To make sure that we do not control this, we create states n states which keep track of the position in the tiling.

6. The tree contains a path in which SPOILER violates a horizontal constraint, which is not immediately followed by an *error* label.

   This NFA works almost the same as the previous. The only difference is that when it finds a violation of the horizontal constraints, it can not immediately accept, but has to check if the next symbol is not equal to 'error', and can then accept.

7. The tree contains a path in which CONSTRUCTOR violates a vertical constraint.

   This automaton is also very similar to the two previous ones. We only have to add n states, that can search for the tile exactly on top of a given tile. For these two tiles, we can check the vertical constraints.

8. The tree contains a path in which SPOILER violates a vertical constraint, which is not immediately followed by an *error* label.

   Again, we just have to extend the previous automaton. When we find a violation of the vertical constraints, we have to check if the next symbol is not equal to 'error'.

We show that a tree t that is not accepted by the first UTA, and which does not contain a path that is accepted by any of the NFAs, encodes a strategy tree for a winning strategy of CONSTRUCTOR.

If t is not accepted by the UTA in item 1, the structure of the tree must be that of a strategy tree. The NFA in item 2 checks that every path that does not end on *error*, has the right length to encode a tiling. The NFAs in item 3 and 4 make sure that for every encoding of a tiling in t, the bottom and top row are equal to $\overline{b}$ and $\overline{t}$, respectively. The last four NFAs check that the horizontal and vertical constraints are respected by CONSTRUCTOR, and that they are respected by SPOILER, or that if they are not respected, this is immediately labeled with *error*. Thus, if t is not accepted by any

of these trees it must encode a strategy tree, in which CONSTRUCTOR has a strategy such that no matter what SPOILER does, SPOILER must place an invalid tile (which is marked by *error*) or a valid corridor tiling is created. In other words, t encodes a strategy tree for a winning strategy of CONSTRUCTOR.

To construct A_2, we first transform the different NFAs into corresponding UTAs using the construction of Lemma 7.5. This gives us a constant number of UTAs. For two UTAs $A_1 = (Q_1, \Sigma, \delta_1, F_1)$, and $A_2 = (Q_2, \Sigma, \delta_2, F_2)$, $A_u = (Q_1 \cup Q_2, \Sigma, \delta_1 \cup \delta_2, F_1 \cup F_2)$ accepts $L(A_1) \cup L(A_2)$, assuming $Q_1 \cap Q_2 = \emptyset$. If that is not the case, it can be achieved by proper renaming. The union of a constant number of UTAs similarly is the union of the state sets, the alphabet, the union of the transition function and the union of the final state sets. We use this technique to compute the union of the UTAs obtained earlier.

As discussed earlier, A_1 accepts all possible trees, and A_2 accepts all trees except those that encode a winning strategy for CONSTRUCTOR in the corridor tiling game on τ. Therefore $L(A_1) = L(A_2)$ if and only if CONSTRUCTOR does not have a winning strategy on τ.

It only remains to show that this reduction can be done in logarithmic space. The construction of A_1, and the constructions of various UTAs and NFAs of A_2, only range over the tiles in O, $\bar{b}$, $\bar{t}$, and n. This requires logarithmic space. The transformation of the NFAs in UTAs only *reads* the NFAs, which are of polynomial size, and thus requires logarithmic space. Finally, taking the union of a constant number of UTAs, of polynomial size, also requires logarithmic space. The problem with this approach is that we have to store intermediate NFAs and UTAs which requires polynomial space. However, we can also describe the whole UTA A_2 immediately, instead of splitting it up into smaller parts. We have split it up because that makes the proof more comprehensible. The construction of this *big* automaton A_2 only ranges over the tiles in O, $\bar{b}$, $\bar{t}$, and n, and can thus be generated in logarithmic space.

We have now shown that the equivalence problem for UTA(RE$(+, \cdot, *)$) is EXPTIME-hard. For the inclusion problem, we do exactly the same reduction, and create the same UTAs A_1 and A_2. Since, A_1 accepts all possible trees, $L(A_1) \subseteq L(A_2)$ if and only if $L(A_1) = L(A_2)$. □

Lemma 7.8. *The equivalence and inclusion problem for UTA($RE(+, \cdot, *, \#)$) and UTA($RE(+, \cdot, *, \&)$) are EXPSPACE-hard.*

Proof. We first prove that the equivalence problem for UTA($RE(+, \cdot, *, \#)$) is EXPSPACE-hard. The proofs for the inclusion problem for UTA($RE(+, \cdot,$

$*, \#)$), and the equivalence and inclusion problem for UTA($RE(+, \cdot, *, \&)$) are analogue.

We reduce the equivalence problem for $RE(+, \cdot, *, \#)$, which is EXPSPACE complete (Theorem 6.2), to the equivalence problem for UTA($RE(+, \cdot, *, \#)$). Given two $RE(+, \cdot, *, \#)$ expressions r_1, r_2 over an alphabet Σ, we construct UTA($RE(+, \cdot, *, \#)$) automata A_1, and A_2, such that $L(r_1) = L(r_2)$ if and only if $L(A_1) = L(A_2)$.

For $i = 1, 2$, we construct $A_i = (Q_i, \Sigma_i, \delta_i, F_i)$ as follows:

- $Q_i = \{q_s\} \cup \Sigma$;
- Assume $\# \notin \Sigma$, $\Sigma_i = \Sigma \cup \{\#\}$;
- $\delta_i(q_s, \#) = r_i$, and for $\sigma \in \Sigma$, $\delta_i(\sigma, \sigma) = \varepsilon$; and
- $F_i = \{q_s\}$.

In this construction, A_1 (resp. A_2) accepts all trees of depth 2, whose root label is #, and for which the string of the children of the root node is defined by r_1 (resp. r_2). It follows that $L(r_1) = L(r_2)$ if and only if $L(A_1) = L(A_2)$.

This construction writes a constant number of elements, and only reads the alphabet of the regular expressions and the regular expressions self. This can al be done in logarithmic space.

For the inclusion problem for UTA($RE(+, \cdot, *, \#)$), we reduce from the inclusion problem for $RE(+, \cdot, *, \#)$, which is also EXPSPACE-complete. We can construct the automata in exactly the same way. The proofs for the equivalence and inclusion problem for UTA($RE(+, \cdot, *, \&)$) are also analogue. The only difference is that we have to reduce from the equivalence and inclusion problem for $RE(+, \cdot, *, \&)$.

□

This concludes the proof of theorem 7.1. □

7.2 Intersection

Theorem 7.9. *The intersection problem for UTA(RE($+, \cdot, *, \#, \&$)) is EXPTIME-complete. It remains EXPTIME-hard for UTA(RE($+, \cdot, *$)), UTA(RE($+, \cdot, *, \#$)), and UTA(RE($+, \cdot, *, \&$)).*

Proof. In Lemma 7.11, we prove that the intersection problem for UTA($RE(+, \cdot, *)$) is EXPTIME-hard. In Lemma 7.12, we prove that the intersection problem for UTA($RE(+, \cdot, *, \#, \&)$) is in EXPTIME. The theorem follows.

Lemma 7.10. *([MNS04]) The intersection problem for* $EDTD^{st}(CHARE((+a), w?, (+a)?, (+a)*))$ *is EXPTIME-hard.*

Lemma 7.11. *The intersection problem for* $UTA(RE(+, \cdot, *))$ *is EXPTIME-hard.*

Proof. We reduce the intersection problem for $\text{EDTD}^{st}(\text{CHARE}((+a), w?, (+a)?, (+a)*))$ (Lemma 7.10) to the intersection problem for $\text{UTA}(RE(+, \cdot, *))$. On input $\text{EDTD}^{st}(\text{CHARE}((+a), w?, (+a)?, (+a)*))$ $D_1, \ldots, D_n$, we must generate $\text{UTA}(RE(+, \cdot, *))$s $A_1, \ldots, A_n$ such that $\bigcap_{i=1}^{n} L(D_i) \neq \emptyset$ if and only if $\bigcap_{i=1}^{n} L(A_i) \neq \emptyset$.

Since every $\text{CHARE}((+a), w?, (+a)?, (+a)*)$-expression is a $RE(+, \cdot, *)$-expression and every single-type EDTD is an EDTD, we know that every $\text{EDTD}^{st}(\text{CHARE}((+a), w?, (+a)?, (+a)*))$ also is an $\text{EDTD}(RE(+, \cdot, *))$. We have already shown that for every $\text{EDTD}(RE(+, \cdot, *))$ D, we can construct an $\text{UTA}(RE(+, \cdot, *))$ A, in logarithmic space, such that $L(A) = L(D)$ (Theorem 5.20).

The reduction constructs for every D_i its equivalent automaton A_i. Since $L(D_i) = L(A_i)$, for every i, we know that $\bigcap_{i=1}^{n} L(D_i) \neq \emptyset$ if and only if $\bigcap_{i=1}^{n} L(A_i) \neq \emptyset$. This reduction uses logarithmic space since the construction of the automata is done in logarithmic space. □

Lemma 7.12. *The intersection problem for* $UTA(RE(+, \cdot, *, \#, \&))$ *is in EXPTIME.*

Proof. Given a number of $\text{UTA}(RE(+, \cdot, *, \#, \&))$s $A_1, \ldots A_n$, $A_i = (Q_i, \Sigma, \delta_i, I_i)$, we have to decide whether $\bigcap_{i=1}^{n} L(A_i) \neq \emptyset$. We give an alternating polynomial space algorithm that tries to guess a tree t that is accepted by every A_i. It accepts if such a tree exists, and rejects if it does not. Since APSPACE = EXPTIME, this concludes the lemma.

The algorithm tries to guess the tree t in a top down manner. For every A_i, we always maintain the current state s_i that A_i is in at the current node. Note that we only have to keep one state and not a set of states, since we only have to proof that a tree is accepted and not that it is rejected by an automaton. Therefore, keeping one state is enough to assure that an accepting run for every A_i on t will be guessed if t is accepted by every A_i.

Guessing the state at a node is not straightforward, since our transition function is defined using regular expressions. However, we can construct an equivalent ENFA for these regular expressions. Note that the alphabet of these ENFAs are the states of the tree automata. By using these ENFA, we can guess the states the A_i are in.

During the computation we use a few variables: $a \in \Sigma$ holds the label at the current node. For every $0 < i \leq n$, $s_i \in Q_i$ contains the state A_i is in at the current node. The string automaton constructed from the transition function of A_i is B_i. The configuration of automaton B_i is denoted by γ_i. The algorithm goes as follows:

1. For $i = 1, \ldots, n$, existentially guess an s_i from I_i.
2. Existentially guess an $a \in \Sigma$.
3. For $i = 1, \ldots, n$, construct an ENFA $B_i = (Q_{B_i}, \Sigma, \delta_{B_i}, q_{B_i}, F_{B_i})$ such that $L(B_i) = L(\delta_i(a, s_i))$.
4. If $\varepsilon \in L(\delta_i(a, s_i)) = L(B_i)$, for every i, ACCEPT.
5. For $i = 1, \ldots, n$, let γ_i be the initial configuration of B_i.
6. For $i = 1, \ldots, n$, guess whether to follow an ε-transition of B_i and update γ_i accordingly.
7. Guess whether to repeat step 6 (GOTO 6) or continue with the algorithm (GOTO 8).
8. For $i = 1, \ldots, n$, guess a non-epsilon transition $w_i \in \delta_i$. If, for any i, there does not exist a configuration η_i such that $\gamma_i \Rightarrow^{w_i} \eta_i$, REJECT. Else, for every i, let $\gamma_i = \eta_i$ and s_i is the symbol read by the transition w_i.
9. For $i = 1, \ldots, n$, guess whether to follow an ε-transition of B_i and update γ_i accordingly.
10. Guess whether to repeat step 9 (GOTO 9) or continue with the algorithm (GOTO 11).
11. If, for every i, γ_i is an accepting configuration of B_i GOTO 11.a, else GOTO 11.b
    (a) Existentially guess whether this node was the last child of its parent (GOTO 2), or if there are more right siblings of the current node (GOTO 11.b)
    (b) Universally guess whether to go to the right sibling of the current node (GOTO 6) or to go to the first child of the current node (GOTO 2).

We explain the algorithm step by step. We start by guessing the state of the root for every automaton (step 1) and the label of the root of our tree (step 2). Then, we construct a string automaton B_i for every automaton A_i based on the state and the label of the root, and the transition function of A_i (step 3). This string automaton accepts all sequences of states, that the children of the root node can have. If, for every i, $\varepsilon \in L(\delta_i(a, s_i))$, we can accept (step 4). In step 5, we set the current configuration of B_i to its initial configuration. Steps 6 and 7 allow every B_i to follow a number of epsilon transitions. After (possibly) following these epsilon transitions, we guess a transition for every B_i. If these transitions are all valid transitions, we update the current configuration of B_i and save the symbol read in s_i. Steps 9 and 10 are identical to steps 6 and 7 and allow the B_i to follow ε-transitions.

Finally, we check whether the configurations γ_i are all accepting configurations (step 11). If that is the case, this node can be the last child of its parent. If we guess that it is the last, we go back to step 2, where we guess the label of the current node, and repeat the entire algorithm. If we guess that it is not the last, we go to step 11.b, where we also arrive when not all γ_i are accepting configurations. There, we have to go to the right sibling (step 6) and go to the first child (goto 2).

If this algorithm accepts, then we have constructed a tree t and an accepting run for every A_i on t. Therefore, $t \in L(A_i)$, for every i.

Suppose that a tree t is accepted by every A_i, and that t is minimal in the sense that we can not construct a tree $t' \neq t$ such that t' is accepted by every A_i, and that for every $u \in Dom(t')$, $lab^{t'}(u) = lab^{t}(u)$. That is, t must have all unnecessary child nodes and subnodes pruned away. Then, there will be a run of our algorithm that (i) guesses t and (ii) guesses the accepting run of A_i on t, for every i. The tree t must be minimal, because our algorithm immediately accepts when possible (see step 4). If the intersection of $L(A_i)$ is non-empty, there exists a minimal tree t accepted by every A_i, and thus our algorithm accepts.

Finally, we show that this algorithm operates in polynomial space. The variables we store, all require no more than polynomial space; a, u_i, and s_i store one element of Σ, Q_{B_i}, and Q_i. The automata B_i, can be constructed in polynomial time, and are stored in polynomial space, and the configurations γ_i can also be stored in polynomial space (Lemma 4.13). In step 2, we have to check if ε is accepted by a constant number of ENFAs. Checking whether ε is accepted by an ENFA B can be done in space polynomial in B (Lemma 4.14). □

This concludes the proof of Theorem 7.9. □

Chapter 8

Conclusion

In this thesis, we have established the complexity of the equivalence, inclusion, and intersection problem for various subclasses of regular expressions and unranked tree automata. For the subclass of regular expressions, where only the standard set of operators $(+, \cdot, *)$ are allowed, the three decision problems are PSPACE-complete. We have shown that the complexity of the equivalence or inclusion problem becomes EXPSPACE when we add the numerical occurrence, shuffle or both operators. The intersection problem remains in PSPACE, even when we add both operators.

We have also extended the class of CHAREs with the numerical occurrence indicator. For the full class of these CHAREs, the complexity of the inclusion and intersection problems remains the same as the complexity of the full set of regular expressions. It is however possible that the equivalence problem for these CHAREs can be solved more efficient. For a much simpler subset of CHAREs, CHARE$(a, a?, a\#)$, the inclusion and intersection problems still remain coNP-hard, and NP-complete. The equivalence problem for CHARE$(a, a?, a\#)$ is tractable. We also found a subset of the CHAREs, CHARE$(a, a\#^{>0})$, for which the three decision problems become tractable.

For unranked tree automata, it was only known that the equivalence and intersection problem for UTA(RE$(+, \cdot, *)$) are EXPTIME-complete. We have shown that the inclusion problem for UTA(RE$(+, \cdot, *)$) is also EXPTIME-complete. As for regular expressions, the complexity of the intersection problem remains the same if we add the numerical occurrence and/or shuffle operators. One would expect that the complexity of the equivalence and inclusion problem would also go one exponential higher, to 2EXPTIME. This, however, is not the case. The addition of one or both operators only takes the equivalence and intersection problem for UTAs to EXPSPACE, which is the same as the corresponding problems for regular expressions.

Using these results and the correlation between the complexities of the

decision problems for regular expressions, and unranked tree automata on one hand, and DTDs, EDTDs, and EDTDst on the other hand, we can establish the complexity of the decision problem for DTDs, EDTDs, and EDTDst. Since these classes are abstractions of DTD, Relax NG, and XML Schema, respectively, this gives us more information about the complexity of the decision problems for these schema languages.

In Table 8.1, we summarize the complexities of the decision problems for DTD, XML Schema, and Relax NG, that can be established using the complexities of the decision problems for regular expressions and unranked tree automata. For example, we know that the complexities of the equivalence problem for a subset of the regular expressions $\mathtt{R}$, DTD($\mathtt{R}$), and EDTDst($\mathtt{R}$) are the same. DTDs allow their regular expressions to use the operators $+, \cdot,$ and $*$. So, the complexity of the equivalence problem for RE$(+, \cdot, *)$, which is PSPACE, carries over to actual DTDs. XML Schema uses the operators $+, \cdot, *, \&$ and $\#$, and the complexity of the equivalence problem for RE$(+, \cdot, *, \#, \&)$ is EXPSPACE. Therefore, the complexity of the equivalence problem for XML Schema also is EXPSPACE.

	equivalence	inclusion	intersection
DTD	PSPACE	PSPACE	PSPACE
XML Schema	EXPSPACE	EXPSPACE	EXPTIME
Relax NG	EXPSPACE	EXPSPACE	EXPTIME

Table 8.1: The complexities of the equivalence, inclusion, and intersection problem for DTD, XML Schema, and Relax NG, where the determinism constraints for DTD and XML Schema are not enforced.

For Relax NG, which is abstracted by EDTDs, we know that UTAs and EDTDs are equivalent. Since Relax NG uses the operators $+, \cdot, *,$ and $\&$, the complexity of the decision problems for UTA(RE$(+, \cdot, *, \&)$) immediately carries over to Relax NG.

The only problem arises for the intersection problem for XML Schema. The complexity of the intersection problem for EDTDst is not the same as that of the class of regular expressions it uses. However, we know that every EDTDst is an EDTD, so the complexity of the intersection problem for UTA(RE$(+, \cdot, *, \#, \&)$), which is EXPTIME, already gives us an upper bound. Furthermore, we know that the intersection problem for EDTDst(CHARE$((+a), w?, (+a)?, (+a)*)$) is already EXPTIME-hard, which gives us an EXPTIME lower bound for XML Schema.

As discussed earlier, we do not enforce the determinism constraints of DTD and XML Schema on our abstractions of these XML schema languages.

This certainly has an influence on the complexity of our decision problems. For example, the equivalence and inclusion problem for DTDs which enforce the determinism constraint are in PTIME, whereas the complexity of these problems for DTD without these constraints is PSPACE. This, however, is not the case in general. The complexity of the intersection problem for DTDs with or without these constraints is PSPACE. Relax NG does not enforce any constraints, so the complexities found there are the *actual* complexities of these problems for Relax NG.

If we look at the results of Table 8.1, we see that none of the decision problems are tractable for the three XML schema languages, without determinism constraints. Among these, the problems can be solved the most efficient for DTDs. This is logical, since DTD is simpler than XML Schema and Relax NG, and does not allow counting or the shuffle operator. We also see that the complexity for the three decision problem is the same for XML Schema and Relax NG. The major difference between our models for XML Schema and Relax NG is that we have incorporated the EDC constraint for XML Schema. However, it turns out that this constraint does not influence the complexity of the decision problems.

These conclusions tell us something about the complexity of the decision problems for the XML schema languages in general. We have also studied CHAREs, because in most practical DTDs and XSDs only CHAREs are used. However, as mentioned earlier, the complexities of the inclusion and intersection problem for the full class of CHAREs are the same as the corresponding problems for the class of regular expressions, which allow the same operators. For the equivalence problem, we have not yet established the exact complexity for CHAREs. We did find subclasses of CHAREs that are tractable. For example, the equivalence problem for CHARE$(a, a?, a\#)$ is in PTIME. It follows that the equivalence problem for DTDs and XSDs which only use regular expressions in CHARE$(a, a?, a\#)$ is also in PTIME. We can make an analogue argumentation for other decision problems and other subclasses of CHAREs or regular expressions.

Bibliography

[BKW98] A. Bruggemann-Klein and D. Wood. One-unambiguous regular languages. *Inf. Comput.*, 140(2):229–253, 1998.

[BMNS05] G. Bex, W. Martens, F. Neven, and T. Schwentick. Expressiveness of xsds: from practice to theory, there and back again. In *WWW*, pages 712–721, 2005.

[BNVdB04] G. Bex, F. Neven, and J. Van den Bussche. DTDs versus XML schema: a practical study. In *WebDB '04: Proceedings of the 7th International Workshop on the Web and Databases*, pages 79–84, New York, NY, USA, 2004. ACM Press.

[BPSM+04] T. Bray, J. Paoli, C. M. Sperberg-McQueen, E. Maler, and F. Yergeau. *Extensible Markup Language (XML) 1.0*, 2004. http://www.w3.org/TR/REC-xml/.

[Chl86] B. S. Chlebus. Domino-tiling games. *J. Comput. Syst. Sci.*, 32(3):374–392, 1986.

[CKS81] A. K. Chandra, D. Kozen, and L. J. Stockmeyer. Alternation. *J. ACM*, 28(1):114–133, 1981.

[CM01] J. Clark and M. Murata. *Relax NG specification*, December 2001. http://www.relaxng.org/spec-20011203.html.

[GJ79] M. R. Garey and D. S. Johnson. *Computers and Intractability : A Guide to the Theory of NP-Completeness*. Series of Books in the Mathematical Sciences. W. H. Freeman, January 1979.

[GKPS05] G. Gottlob, C. Koch, R. Pichler, and L. Segoufin. The complexity of xpath query evaluation and xml typing. *J. ACM*, 52(2):284–335, 2005.

[JS01] J. Jedrzejowicz and A. Szepietowski. Shuffle languages are in P. *Theor. Comput. Sci.*, 250(1-2):31–53, 2001.

[Koz77] D. Kozen. Lower bounds for natural proof systems. In *Proceedings of the 18th Symposium on Foundations of Computer Science*, pages 254–266, Los Alamitos, CA, October 1977. IEEE Computer Society Press.

[KT03] P. Kilpeläinen and R. Tuhkanen. Regular expressions with numerical occurrence indicators - preliminary results. In *Proc. of the Eighth Symposium on Programming Languages and Software Tools*, pages 163–173, 2003.

[LP97] H. R. Lewis and C. H. Papadimitriou. *Elements of the Theory of Computation*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1997.

[Man01] M. Mani. Keeping chess alive - do we need 1-unambiguous content models? *Extreme Markup Languages*, 2001.

[Mar06] W. Martens. *Static Analysis of XML Transformation and Schema Languages*. PhD thesis, Universiteit Hasselt, 2006.

[MLMK05] M. Murata, D. Lee, M. Mani, and K. Kawaguchi. Taxonomy of XML schema languages using formal language theory. *ACM Trans. Internet Techn.*, 5(4):660–704, 2005.

[MNS04] W. Martens, F. Neven, and T. Schwentick. Complexity of decision problems for simple regular expressions. In *MFCS*, pages 889–900, 2004.

[MS94] A. J. Mayer and L. J. Stockmeyer. The complexity of word problems - this time with interleaving. *Inf. Comput.*, 115(2):293–311, 1994.

[Nev02] F. Neven. Automata theory for XML researchers. *SIGMOD Record*, 31(3):39–46, 2002.

[Pap94] C. H. Papadimitriou. *Computational Complexity*. Addison-Wesley, 1994.

[PV00] Y. Papakonstantinou and V. Vianu. DTD Inference for Views of XML Data. In *Proceedings of the Nineteenth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, pages 35–46, Dallas, Texas, 2000.

[Sei90] H. Seidl. Deciding equivalence of finite tree automata. *SIAM J. Comput.*, 19(3):424–437, 1990.

[Sei94] H. Seidl. Haskell overloading is dexptime-complete. *Inf. Process. Lett.*, 52(2):57–60, 1994.

[Sip96] M. Sipser. *Introduction to the Theory of Computation*. International Thomson Publishing, 1996.

[SM73] L. J. Stockmeyer and A. R. Meyer. Word problems requiring exponential time(preliminary report). In *STOC '73: Proceedings of the fifth annual ACM symposium on Theory of computing*, pages 1–9, New York, NY, USA, 1973. ACM Press.

[SM03] C. M. Sperberg-McQueen. XML schema 1.0: A language for document grammars. In *XML 2003 - Conference Proceedings*, 2003.

[SMT05] C. M. Sperberg-McQueen and H. Thompson. *XML Schema*, 2005. http://www.w3.org/XML/Schema.

[Suc02] D. Suciu. Typechecking for semistructured data. In *DBPL '01: Revised Papers from the 8th International Workshop on Database Programming Languages*, pages 1–20, London, UK, 2002. Springer-Verlag.

[Tha73] J. W. Thatcher. Tree automata: An informal survey. In A. V. Aho, editor, *Currents in the Theory of Computing*, chapter 4, pages 143–172. Prentice-Hall, Englewood Cliffs, NJ, 1973.

[vdV02] E. van der Vlist. *XML Schema*. O'Reilly, 2002.

[vEB97] P. van Emde Boas. The convenience of tilings. *Complexity, Logic and Recursion Theory, Lecture Notes in Pure and Applied Mathematics*, 187:331–363, 1997.

Samenvatting

XML (eXtensible Markup Language) ([BPSM+04]) is een W3C standaard voor het uitwisselen van gestructureerde documenten en data over het internet. Het is een formaat dat zeer flexibel is omdat het user-defined *tags* toelaat. Bovendien kunnen zowel mensen als computers de inhoud van een XML document gemakkelijk interpreteren. Daarom is XML de laatste jaren de standaard geworden voor het uitwisselen van data over het internet. Een voorbeeld van een XML document wordt getoond in Figuur 8.1.

XML schema's laten gebruikers toe om een eigen formaat voor XML documenten te definiëren. Een XML schema beschrijft de tags die in het XML document gebruikt mogen worden, en de structuur die het XML document moet hebben. Gebruik maken van XML schema's heeft veel voordelen. Een voorbeeld hiervan is het probleem van data integratie. Stel dat er twee databronnen op het internet zijn, waarin de data in XML documenten is opgeslaan. Bovendien heeft elke databron een XML schema dat al zijn XML documenten definieert. Dan is het mogelijk om deze bronnen in één database te integreren door enkel de XML schema's te bestuderen, en niet elk document afzonderlijk.

De meest gebruikte en verspreide XML schema talen zijn *Document Type Definitions* (DTDs) ([BPSM+04]), *XML Schema* ([SMT05]), en *Relax NG* ([CM01]). De eerste XML schema taal die echt populair is geworden, is DTD. DTD is een simpele taal die reguliere expressies gebruikt om de structuur van XML documenten te definiëren. Een voorbeeld van een DTD die het XML document uit Figuur 8.1 definieerd, wordt getoond in Figuur 8.2.

DTDs zijn de standaard XML schema taal geworden, maar de mogelijkheden van DTDs zijn eerder beperkt. Daarom zijn er een aantal nieuwe XML schema talen ontwikkeld. De meest populaire van deze schema talen is XML Schema. XML Schema heeft een XML gebaseerde syntax. Dit wil zeggen dat een *XML Schema Definition* (XSD) zelf ook een XML document is. XML Schema bevat ook een systeem om types toe te kennen aan elementen, en een groot aantal andere mogelijkheden, die DTD niet heeft. Eén van deze mogelijkheden is dat je kan uitdrukken dat een aantal elementen in

```
<cd>
    <song>
        <title>Susan's house</title>
        <length>223</length>
        <singlesSold>55000</singlesSold>
    </song>
    <song>
        <title>Beautiful freak</title>
        <length>213</length>
        <singlesSold>100000</singlesSold>
    </song>
    <song>
        <title>Flower</title>
        <length>227</length>
    </song>
</cd>
```

Figure 8.1: Een XML document dat een deel van het album 'Beautiful Freak' van Eels beschrijft. Voor ieder liedje is de titel en de lengte (in seconden) gegeven. Indien het lied een single is geweest, is ook het aantal verkochte singles gegeven. De waarden zijn willekeurig gekozen.

```
<!DOCTYPE cd [
  <!ELEMENT cd (song*)>
  <!ELEMENT song (title,length,singlesSold?)>
  <!ELEMENT title (#PCDATA)>
  <!ELEMENT length (#PCDATA)>
  <!ELEMENT singlesSold (#PCDATA)>
]>
```

Figure 8.2: Een DTD die het XML document uit Figuur 8.1 definieert.

een willekeurige volgorde mogen voorkomen. Dit kunnen we doen door deze elementen te omringen met de *all* tag. Indien we hetzelfde willen uitdrukken in een DTD, moeten we alle mogelijke permutaties van deze elementen opsommen. Dit geeft $n!$ mogelijkheden voor n elementen. Een andere extra functie van XML Schema is dat het toelaat om het minimum en maximum aantal keren dat een term mag voorkomen aan te geven. Dit kan je doen door de *minoccurs* en *maxoccurs* attributen van een element te specifiëren. Het is opnieuw mogelijk om dit in een DTD te doen, maar dan moeten we dat element *maxoccurs* keer herhalen. In Figuur 8.3 tonen we een XSD die het XML document van Figuur 8.1 definieert. We gebruiken hier de *all* tag om aan te geven dat de elementen *title*, *length*, en *singlesSold* in een willekeurige volgorde mogen voorkomen. De *minoccurs* en *maxoccurs* attributen geven aan dat een cd minimaal één liedje en maximaal twintig liedjes kan bevatten.

Een andere uitbreiding van DTD is Relax NG. Dit is een zeer elegante en krachtige XML schema taal, maar ze is niet zo populair als XML Schema. Relax NG heeft een XML gebaseerde syntax, en een eenvoudigere equivalente *compact* syntax. Net zoals XML Schema heeft het een typing systeem voor zijn elementen. Als een alternatief voor de *all* tag van XML Schema, laat het de &-operator toe in zijn reguliere expressies. Deze binaire operator heet de shuffle of interleave operator, en laat toe om de woorden die door zijn twee termen geaccepteerd worden, door elkaar te schrijven, of te *shufflen*. Relax NG heeft geen alternatief voor de *minoccurs* en *maxoccurs* attributen van XML schema. In Figuur 8.4 tonen we een deel van een Relax NG document, dat het element *song* definieert zoals het is beschreven door de XSD in Figuur 8.3. We gebruiken hier de shuffle operator om de *all* tag van XML Schema te vervangen.

In dit werk zijn we geïnteresseerd in de optimalisatie van XML schema talen. Een voorbeeld van een optimalisatie probleem is het minimaliseren van XML schema's. Een geminimaliseerd XML schema laat ons toe om document validatie efficiënter te doen, en ook andere tests op het schema kunnen sneller

```
<schema>
    <element name="cd" type="cdType">

    <xs:element name="single">
        <xs:complexType>
            <xs:sequence>
                <xs:all>
                    <xs:element name="title" type="xs:string"/>
                    <xs:element name="length" type="xs:integer"/>
                    <xs:element name="singlesSold" type="xs:integer"/>
                </xs:all>
            </xs:sequence>
        </xs:complexType>
    </xs:element>

    <xs:element name="regular">
        <xs:complexType>
            <xs:sequence>
                <xs:all>
                    <xs:element name="title" type="xs:string"/>
                    <xs:element name="length" type="xs:integer"/>
                </xs:all>
            </xs:sequence>
        </xs:complexType>
    </xs:element>

    <xs:element name="cdType">
        <xs:complexType>
            <xs:sequence>
                <xs:choice minoccurs = "1" maxoccurs = "20">
                    <xs:element name="song" type="single"/>
                    <xs:element name="song" type="regular"/>
                </xs:choice>
            </xs:sequence>
        </xs:complexType>
    </xs:element>
</schema>
```

Figure 8.3: Een XSD die het XML document van Figuur 8.1 definieert, maar de EDC constraint overtreedt.

```
element song {
    element title { text }
    & element length { xsd:integer }
    & element singlesSold { xsd:integer }?
}
```

Figure 8.4: Een beschrijving van het element *song* in Relax NG, in compacte syntax, zoals gedefinieerd door de XSD in figuur 8.3.

uitgevoerd worden. Om een schema te minimaliseren, kunnen we een kleiner schema opstellen, en controleren of dit kleiner schema nog steeds dezelfde verzameling van XML documenten definieert als het originele XML schema. Het probleem, waarbij we controleren of twee schema's dezelfde verzameling XML documenten definiëren, noemen we het equivalentieprobleem.

Analoog aan het equivalentieprobleem, definiëren we ook het inclusie- en intersectieprobleem.

- Equivalentie: Gegeven twee schema's D_1 en D_2. Definiëren D_1 en D_2 dezelfde verzameling XML documenten?
- Inclusie: Gegeven twee schema's D_1 en D_2. Wordt elk XML document dat gedefinieerd wordt door D_1, ook gedefinieerd door D_2?
- Intersectie: Gegeven n schema's $D_1, \ldots, D_n$. Bestaat er een XML document dat door elke D_i gedefinieerd wordt, $i = 1, \ldots, n$?

Deze beslissingsproblemen zijn de bouwstenen van de meeste optimalisatieproblemen voor XML schema's. Daarom is het belangrijk om de exacte complixeteit van deze problemen te bepalen voor de verschillende XML schema talen.

Om de complexiteit van deze problemen te bestuderen, maken we een abstractie van XML documenten en van de verschillende XML schema talen. XML documenten worden geabstraheerd door bomen. Figuur 8.5 illustreert de overeenkomstige boom voor het XML document uit Figuur 8.1.

We gebruiken *uitgebreide context vrije grammatica's*, om een abstractie te maken van DTDs. Deze uitgebreide context vrije grammatica's gebruiken reguliere expressies om hun inhoud te beschrijven, en we zullen deze abstracties ook DTDs noemen.

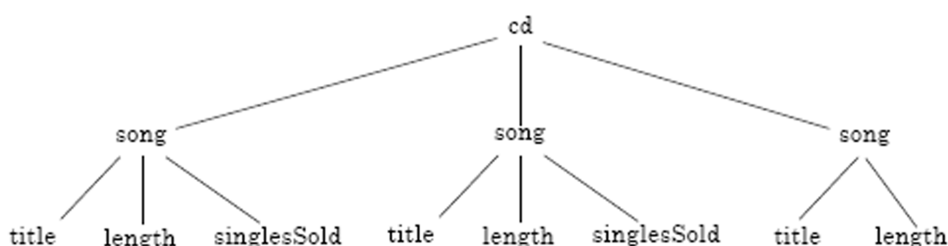

Figure 8.5: Het XML document uit Figuur 8.1 voorgesteld als een boom.

Example 8.1. *We kunnen een abstractie maken van de DTD in Figuur 8.2 met behulp van de volgende DTD:*

$$\begin{aligned} cd &\rightarrow song^* \\ song &\rightarrow title\ length\ singlesSold? \end{aligned}$$

Eén van de grote verschillen tussen DTD enerzijds, en XML Schema en Relax NG anderzijds, is de mogelijkheid om types aan elementen toe te kennen. Daarom breiden we DTDs uit door ook hieraan types toe te voegen. Dit geeft ons *extended* DTDs (EDTDs) ([PV00]). Deze EDTDs zijn een goede abstractie van Relax NG, maar XML Schema legt een aantal beperkingen op die Relax NG niet heeft. Daarom zijn deze EDTDs te krachtig om XML Schema voor te stellen. Eén van deze beperkingen is *element declarations consistent* (EDC). EDC drukt uit dat in een reguliere expressie, er geen elementen met dezelfde naam, maar met een vershillend type mogen voorkomen. We kunnen de kracht van EDTDs beperken tot *single-type* EDTDs (EDTD^{st}) ([BMNS05],[MLMK05]). Deze EDTD^{st}s beschrijven precies de XML schema talen met de EDC beperking. Een EDTD^{st} laat niet toe dat een reguliere expressie twee elementen met dezelfde naam, maar met verschillende types bevat. We gebruiken EDTD^{st}s als een abstractie voor XML Schema.

Example 8.2. *We kunnen een abstractie maken van de XSD in Figuur 8.3 met behulp van de volgende EDTD:*

$$\begin{aligned} cd &\rightarrow (song^1 + song^2)^{1..20} \\ song^1 &\rightarrow title\ \&\ length\ \&\ singlesSold \\ song^2 &\rightarrow title\ \&\ length \end{aligned}$$

Hier stelt $song^1$ *de liedjes voor die een single zijn geweest, en* $song^2$ *stelt de gewone liedjes voor. Deze EDTD is geen* $EDTD^{st}$ *aangezien* $song^1$ *en* $song^2$ *voorkomen in dezelfde expressie. Dit komt omdat de XSD in Figuur 8.3 de EDC constraint overtreedt. We kunnen dit oplossen door de XSD te herschrijven.* □

We moeten wel opmerken dat DTD en XML schema een beperking opleggen die zegt dat de gebruikte reguliere expressies deterministisch of one-unambiguous moeten zijn. Dit wordt uitgedrukt door de *Deterministic Content Models* voor DTD en de *Unique Particle Attribution* constraint voor XML Schema. We leggen deze beperkingen niet op aan onze modellen voor DTD (DTD) en XML Schema (EDTD^{st}), omdat er al vaker discussie is geweest over deze beperkingen in de XML gemeenschap (zie bijvoorbeeld, pagina 98 van [vdV02], [Man01], of [SM03]). Daarom is het interessant om te zien wat de complexiteit van onze beslissingsproblemen voor DTD en XML Schema is, wanneer deze beperkingen niet worden opgelegd.

DTD, XML Schema, en Relax NG, en hun abstracties, gebruiken allemaal (deelverzamelingen van) reguliere expressies om de structuur van XML documenten te beschrijven. In deze reguliere expressies worden er een aantal verschillende operatoren gebruikt. DTDs gebruiken de gebruikelijke verzameling van operatoren $+, \cdot$, en $*$. XML Schema voegt een aantal mogelijkheden toe aan DTD. We hebben reeds gezien dat de *all* tag kan vertaald worden met behulp van de shuffle operator (&). De *mincount* en *maxcount* attributen kunnen we vertalen met behulp van de *numerical occurrence operator*. Het gebruik van deze operator wordt geïllustreerd in Figuur 8.2, en geeft in superscript het minimum en maximum aantal voorkomens aan. Hij wordt hier gebruikt om aan te geven dat een cd minimaal één liedje, en maximaal twintig liedjes kan bevatten. We stellen deze operator voor als #. We kunnen nu deelverzamelingen van de reguliere expressies definiëren, door te specifiëren welke operators wel, en welke niet gebruikt mogen worden in de reguliere expressies.

Het blijkt nu dat de complexiteit van de beslissingsproblemen (equivalentie, inclusie, en intersectie) voor DTDs, en EDTD^{st}s, zeer verwant is met de complexiteit van dezelfde beslissingsproblemen voor dezelfde deelverzameling van reguliere expressies die ze gebruiken ([MNS04]). We definiëren ook boomautomaten (UTA, van het engelse unranked tree automata), die ook (deelverzamelingen van) reguliere expressies gebruiken. Deze boomautomaten zijn even krachtig als EDTDs. Daarom is de complexiteit van de beslissingsproblemen voor EDTDs en UTAs steeds hetzelfde. Door deze sterke overeenkomsten tussen DTDs, EDTDs, en EDTD^{st}s enerzijds, en reguliere expressies en UTAs anderzijds, zullen we de complexiteit van de beslissingsproblemen voor verschillende deelverzamelingen van reguliere expressies en UTAs onderzoeken.

We weten reeds dat we iedere "standaard" $\text{RE}(+, \cdot, *)$-expressie kunnen vertalen naar een equivalentie NFA, met polynomiale grootte. We kunnen dan deze NFAs gebruiken om bovengrenzen voor de beslissingproblemen voor $\text{RE}(+, \cdot, *)$ te bepalen. Wanneer we echter een willekeurige $\text{RE}(+, \cdot, *, \&, \#)$-

expressie in een equivalente NFA willen vertalen, kan het zijn dat deze NFA een dubbel-exponentiële grootte heeft. Daarom introduceren we een nieuwe soort automaten, *extended* NFAs (ENFAs). Deze ENFAs zijn een uitbreiding van gewone NFAs, en laten ons toe om iedere RE$(+, \cdot, *, \&, \#)$-expressie in een equivalente ENFA van polynomiale grootte te vertalen. We gebruiken deze ENFAs vaak om bovengrenzen voor beslissingsproblemen voor RE$(+, \cdot, *, \&, \#)$ te bepalen.

We beschouwen eerst deelverzamelingen van reguliere expressies, die gedefinieerd worden door het toelaten van een aantal operatoren in de reguliere expressies. In Tabel 8.2 geven we een overzicht van de (reeds bekende en nieuwe) resultaten.

	equivalentie	inclusie	intersectie
RE$(+, \cdot, *)$	PSPACE [SM73]	PSPACE [SM73]	PSPACE [Koz77]
RE$(+, \cdot, *, \#)$	EXPSPACE	EXPSPACE	PSPACE
RE$(+, \cdot, *, \&)$	EXPSPACE [MS94]	EXPSPACE [MS94]	PSPACE
RE$(+, \cdot, *, \#, \&)$	EXPSPACE	EXPSPACE	PSPACE

Table 8.2: Complexiteit voor problemen met reguliere expressies. Alle resultaten zijn compleetheidsresultaten. Voor resultaten die reeds bekend waren in de literatuur, is de referentie toegevoegd.

We zien dat alle drie de problemen PSPACE-compleet zijn voor RE$(+, \cdot, *)$. Wanneer we bij het equivalentie- of inclusieprobleem, de shuffle of numerical occurrence operator toevoegen, worden deze problemen EXPSPACE-compleet. Indien ze beiden worden toegevoegd, blijven we wel in EXPSPACE. Je zou verwachten dat dezelfde trend bestaat voor het intersectieprobleem. Dit is echter niet waar. Het intersectieprobleem blijft in PSPACE, zelfs wanneer we beide operatoren toevoegen.

Deze deelverzamelingen van reguliere expressies bevatten allen zeer complexe expressies. Het blijkt echter dat de reguliere expressies die gebruikt worden in echte DTDs en XSDs, meestal zeer simpel zijn. Bex, Neven, en Van den Bussche hebben hier een studie over gedaan ([BNVdB04]), en het blijkt dat meer dan negentig procent van de reguliere expressies die voorkomen in echte DTDs en XSDs, uitdrukkingen zijn van de vorm $e_1 \cdots e_n$, waarbij iedere e_i een factor is van de vorm $(x_1 + \cdots + x_n)$, mogelijks uitgebreid met de *, +, of ? operatoren, en iedere x_i een string is. Martens, Neven, en Schwentick hebben deze klasse van reguliere expressies gedefinieerd als CHAREs, en hebben de complexiteit van een aantal deelverzamelingen van deze CHAREs bestudeerd ([MNS04]). In dit werk, breiden we deze CHAREs uit met de numerical occurrence operator. In Tabel 8.3, geven we een overzicht van een

aantal resultaten uit [MNS04], en de resultaten van dit werk.

	equivalentie	inclusie	intersectie
CHARE($S\backslash\{\#\}$)	in PSPACE *	PSPACE *	PSPACE *
CHARE(S)	in EXPSPACE	EXPSPACE	PSPACE
CHARE($a, a?$)	in PTIME *	coNP *	NP *
CHARE($a, a*$)	in PTIME *	coNP *	NP *
CHARE($a, a?, a\#$)	in PTIME	coNP-hard, in EXPSPACE	NP
CHARE($a, a\#^{>0}$)	in PTIME	in PTIME	in PTIME

Table 8.3: Complexiteit voor beslissingsproblem voor CHAREs. Alle resultaten zijn compleetheidsresultaten, tenzij het anders vermeld is. Resultaten gemarkeerd met een * zijn gevonden door Martens, Neven, and Schwentick ([MNS04]).

De klasse CHARE(S) bevat alle mogelijke CHAREs, uitgebreid met de numerical occurrence operator. We zien dat voor het inclusie- en intersectieprobleem, de complexiteit dezelfde is als deze van RE($+, \cdot, *, \#$). Het is echter wel mogelijk dat het equivalentieprobleem efficiënter kan opgelost worden. We beschouwen ook veel eenvoudigere deelverzamelingen van deze CHAREs, CHARE($a, a?, a\#$), en CHARE($a, a\#^{>0}$). Voor CHARE($a, a?, a\#$), zijn het inclusie- en intersectieprobleem nog steeds coNP-hard en NP-compleet. Voor CHARE($a, a\#^{>0}$) zijn alle drie de problemen in PTIME.

Ten slotte bestuderen we de complexiteit van de beslissingsproblemen voor UTAs. De complexiteit van deze problemen hangt af van de deelverzameling van de reguliere expressies die in de transitiefunctie gebruikt mogen worden. De klasse UTA(RE($+, \cdot, *, \&$)) bevat bijvoorbeeld alle UTAs, die enkel de drie standaard operatoren in de reguliere expressies van hun transitiefunctie gebruiken. In Tabel 8.4 geven we een overzicht van de resultaten voor UTAs.

We zien dat de drie problemen voor UTA(RE($+, \cdot, *$)) EXPTIME-compleet zijn. Bovendien blijft de complexiteit van het intersectieprobleem, net zoals bij reguliere expressies, hetzelfde wanneer we de twee extra operatoren toevoegen. In vergelijking met de overeenkomstige problemen voor reguliere expressies, gaan we dus van plaats- naar tijdcomplexiteit, maar gaan we ook exponentieel hoger. Wanneer we bij het equivalentie- en inclusieprobleem de nieuwe operatoren toevoegen, worden deze problemen maar EXPSPACE-compleet. Hier blijft deze trend dus niet behouden. Sterker nog, de complexiteit van deze problemen is dezelfde als deze van de overeenkomstige

	equivalentie	inclusie	intersectie
UTA(RE($+, \cdot, *$))	EXPTIME [Sei90]	EXPTIME	EXPTIME [Sei94]
UTA(RE($+, \cdot, *, \#$))	EXPSPACE	EXPSPACE	EXPTIME
UTA(RE($+, \cdot, *, \&$))	EXPSPACE	EXPSPACE	EXPTIME
UTA(RE($+, \cdot, *, \#, \&$))	EXPSPACE	EXPSPACE	EXPTIME

Table 8.4: Complexiteit voor problemen met UTAs. Alle resultaten zijn compleetheidsresultaten. De complexiteiten van het equivalentie- en intersectieprobleem voor UTA(RE($+, \cdot, *$)) zijn gevonden door Seidl.

problemen voor reguliere expressies.

We hebben de beslissingsproblemen voor de reguliere expressies en UTAs bestudeerd, omdat dit ons iets meer kon vertellen over dezelfde problemen voor DTD, XML Schema, en Relax NG. Op basis van deze resultaten kunnen we Tabel 8.5 samenstellen. Deze tabel geeft ons voor de verschillende XML schema talen de complexiteit van de beslissingsproblemen. Merk wel op dat deze resultaten geen rekening houden met de beperkingen die zeggen dat de reguliere expressies in DTDs en XSDs deterministisch of one-unambiguous moeten zijn. Deze beperkingen zorgen ervoor dat sommige, maar zeker niet alle, problemen efficiënter kunnen opgelost worden.

	equivalentie	inclusie	intersectie
DTD	PSPACE	PSPACE	PSPACE
XML Schema	EXPSPACE	EXPSPACE	EXPTIME
Relax NG	EXPSPACE	EXPSPACE	EXPTIME

Table 8.5: De complexiteit van de verschillende beslissingsproblemen, voor DTD, XML Schema, en Relax NG, waarbij de beperkingen voor deterministische reguliere expressies voor DTD en XML Schema niet zijn opgelegd.

Indien we naar de resultaten van Tabel 8.5 kijken, zien we dat geen enkel van deze problemen voor geen enkele XML schema taal efficiënt oplosbaar is. Indien we de drie schema talen vergelijken, zijn de problemen het meest efficiënt oplosbaar voor DTD. Dit is logisch aangezien DTD het meest simpel is, en geen numerical occurrence of shuffle operator toelaat. We zien bovendien dat de complexiteit voor de verschillende problemen voor XML Schema en Relax NG steeds hetzelfde is. Hieruit volgt dat de EDC beperking, die we aan ons model voor XML Schema opgelegd hebben, geen invloed heeft op de complexiteit van deze problemen.

Deze conclusies vertellen ons iets over de complexiteit van de beslissings-

problemen voor de XML schema talen in het algemeen. We hebben ook CHAREs bestudeerd, omdat de meeste reguliere expressies die in de praktijk gebruikt worden, CHAREs zijn. Op basis van de resultaten voor CHAREs kunnen we weer een aantal resultaten voor DTD en XML Schema afleiden. Bijvoorbeeld, uit het feit dat het equivalentieprobleem voor CHARE($a, a?, a\#$) in PTIME is, volgt dat het equivalentieprobleem voor DTD en XML Schema, waarbij enkel CHARE($a, a?, a\#$)-expressies mogen gebruikt worden, ook in PTIME is.