Complexity of Decision Problems for XML Schemas and Chain Regular Expressions∗

Wim Martens  Frank Neven  Thomas Schwentick

October 30, 2006

Abstract

We study the complexity of the inclusion, equivalence, and intersection problem for XML schemas occurring in practice. These schemas make use of regular expressions with a very simple structure: they basically consist of the concatenation of factors, where each factor is a disjunction of strings, possibly extended with “∗”, “+”, or “?”. We refer to these as CHAin Regular Expressions (CHAReEs). We obtain lower and upper bounds for various fragments of CHAReEs and also consider additional determinism and occurrence constraints. For the equivalence problem, we only prove an initial tractability result, leaving the complexity of more general cases open. We relate the above to optimization of XML schema languages by showing that all our lower and upper bounds for the inclusion and equivalence problem carry over to the corresponding decision problems for extended context-free grammars, and to single-type and restrained competition tree grammars. The latter form abstractions of Document Type Definitions (DTDs), XML Schema definitions (XSDs) and the class of one-pass preorder typeable XML schemas, respectively. For the intersection problem, we show that obtained complexities only carry over to DTDs.

1 Introduction

XML (eXtensible Markup Language) has become the standard data exchange format for the World Wide Web [ABS99]. Within a community, parties usually do not exchange arbitrary XML documents, but only those conforming to a predefined format. Such a format is usually called a schema and is specified in some schema language. The presence of a schema accompanying an XML document has many advantages: it allows for automation and optimization of search, integration, and processing of XML data (cf., e.g., [BFG05, DFS99, KSSS04, MFK01, NS03, WLY +03]). Furthermore, for typechecking or type inference of XML transformations [HP03, MN05, MSV03, PV00], schema information is even

crucial. The following standard optimization problems for schemas are among the basic building blocks for many of the algorithms for the above mentioned problems:

- **INCLUSION**: Given two schemas $E$ and $E'$, is every XML document in $E$ also defined by $E'$?
- **EQUIVALENCE**: Given two schemas $E$ and $E'$, do $E$ and $E'$ define the same set of XML documents?
- **INTERSECTION**: Given $E_1, \ldots, E_n$, do they define a common XML document?

It is therefore important to establish the exact complexity of these problems.

First, we discuss the schema languages we are interested in. Although many XML schema languages have been proposed, Document Type Definitions (DTDs) [BPSM+04] and XML Schema Definitions (XSDs) [SMT05] are the most prevalent ones. Generally, these languages are abstracted by extended context-free grammars, unranked tree automata, or extended grammars, respectively [Nev02, Via01]. Extended context-free grammars are context-free grammars with regular expressions as right-hand sides of rules, while the unranked tree automata are a natural extension of classical tree automata to trees where nodes can have an unbounded number of children [BKMW01]. Extended grammars or also called extended DTDs (EDTDs)\(^1\) are a grammar-based alternative to unranked tree automata [PV00]. As grammars and tree automata have already been studied in depth for many decades, it is not surprising that the complexity of the above mentioned decision problems is already known. Indeed, in the case of DTDs, the problems reduce to their counterparts for regular expressions: all three problems are PSPACE-complete [Koz77, SM73]. For tree automata, they are well-known to be EXPTIME-complete [Sei90, Sei94].

Unfortunately, these complexity results result do not tell us much about the hardness of optimization of actual XML schemas:

1. In contrast to what is generally assumed, XSDs do not correspond to the complete class of general unranked tree automata or EDTDs, but to a strict subclass of EDTDs, namely, single-type EDTDs [MLMK05, MNS05].

2. Practical DTDs and XSDs are usually much simpler than the regular expressions and tree automata needed for the classical PSPACE- and EXPTIME-hardness proofs. Actually, a study by Bex, Neven, and Van den Bussche [BNV04] reveals that more than ninety percent of the regular expressions occurring in practical DTDs and XSDs are CHAIN Regular Expressions (CHAREs), that is, expressions $e_1 \cdots e_n$, where every $e_i$ is a factor of the form $(w_1 + \cdots + w_m)$ — possibly extended with Kleene-star, plus or

\(^1\)Papakonstantinou and Vianu used the term *specialized DTD* as types *specialize* tags. We prefer the term *extended DTD* as it expresses more clearly that the power of the schemas is amplified.
Table 1: Possible factors in chain regular expressions and how they are denoted
\((a, a_i \in \Sigma, w, w_i \in \Sigma^+)\).

<table>
<thead>
<tr>
<th>Factor</th>
<th>Abbr.</th>
</tr>
</thead>
<tbody>
<tr>
<td>(a)</td>
<td>(a)</td>
</tr>
<tr>
<td>(a^*)</td>
<td>(a^*)</td>
</tr>
<tr>
<td>(a^+)</td>
<td>(a^+)</td>
</tr>
<tr>
<td>(a?)</td>
<td>(a?)</td>
</tr>
<tr>
<td>(w^*)</td>
<td>(w^*)</td>
</tr>
<tr>
<td>(w^+)</td>
<td>(w^+)</td>
</tr>
<tr>
<td>(w?)</td>
<td>(w?)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Factor</th>
<th>Abbr.</th>
</tr>
</thead>
<tbody>
<tr>
<td>((a_1 + \cdots + a_n))</td>
<td>((+a))</td>
</tr>
<tr>
<td>((a_1 + \cdots + a_n)^*)</td>
<td>((+a)^*)</td>
</tr>
<tr>
<td>((a_1 + \cdots + a_n)^+)</td>
<td>((+a)^+)</td>
</tr>
<tr>
<td>((a_1 + \cdots + a_n)?)</td>
<td>((+a)?)</td>
</tr>
<tr>
<td>((a_1^* + \cdots + a_n^*))</td>
<td>((+a^*))</td>
</tr>
<tr>
<td>((w_1^* + \cdots + w_n^*))</td>
<td>((+w^*))</td>
</tr>
</tbody>
</table>

In the present paper, we therefore revisit the complexity of INCLUSION, EQUIVALENCE, and INTERSECTION for DTDs and single-type EDTDs with CHAREs. In addition, we also consider RESTRAINED COMPETITION EDTDs [MLMK05, MNS05], which is a class of schema languages that lies strictly between single-type EDTDs and general EDTDs in terms of expressive power. They have been proposed by Murata et al., and have been shown to capture the class of EDTDs that can be typed in a streaming fashion [MNS05, BMNS05]. That is, when reading an XML document as a SAX-stream, they allow to determine the type of any element when its opening tag is met (that is, one-pass preorder typing). Clearly, complexity lower bounds for INCLUSION, EQUIVALENCE, or INTERSECTION for a class of regular expressions \(R\) imply lower bounds for the corresponding decision problems for DTDs, single-type EDTDs, and restrained competition EDTDs with right-hand sides in \(R\). Interestingly, we show that for INCLUSION and EQUIVALENCE, the complexity upper bounds for the string case also carry over to DTDs, single-type EDTDs, and restrained competition EDTDs. For intersection, the latter still holds for DTDs, but not for single-type or restrained competition EDTDs. So, in many cases, it suffices to restrict attention to the complexity of CHAREs to derive complexity bounds for XML schema languages.

Before we give an overview of our complexity results regarding CHAREs, we briefly discuss the determinism constraint: the XML specification requires DTD content models to be deterministic because of compatibility with SGML (Section 3.2.1 of [BPSM+04]). In XML Schema, this determinism constraint is referred to as the unique particle attribution constraint (Section 3.8.6 of [SMT05]). BrüggeMann-Klein and Wood [BKW98] formalized the regular expressions adhering to this constraint as the one-unambiguous regular expressions. A relevant property is that such expressions can be translated in polynomial time to an equivalent deterministic finite state machine. Hence, it immediately fol-

---

2We refer to Section 2 for a detailed definition of chain regular expressions.
allows that inclusion and equivalence for such regular expressions, and hence also for practical DTDs, are in PTIME. In contrast, we show in Theorem 6.4 that intersection of one-unambiguous regular expressions remains PSPACE-hard (even when every symbol can occur at most three times). Nevertheless, we think it is important to also study the complexity of CHAREs without the determinism constraint as there has been quite some debate in the XML community about the restriction to one-unambiguous regular expressions (cf., for example, page 98 of [vdV02] and [Man01, SM03]). Indeed, several people want to abandon the notion as its only reason for existence is to ensure compatibility with SGML parsers and, furthermore, because it is not a transparent one for the average user which is witnessed by several practical studies [BNV04, Cho02] that found a number of non-deterministic content models in actual DTDs. In fact, Clarke and Murata already abandoned the notion in their Relax NG specification [vdV03], the most serious competitor for XML Schema. Another reason to study CHAREs without the determinism constraint is that they are included in navigational queries expressed by caterpillar expressions [BKW00], $\mathcal{X}_{\text{reg}}^{\text{CPath}}$ and $\mathcal{X}_{\text{reg}}$ [Mar04], or regular path queries [CGGLV03]. Hence, lower bounds for optimization problems for CHAREs imply lower bound for optimization problems for navigational queries. Hence, it is relevant to study the broader class of possibly non-deterministic but simple and practical regular expressions.

Our results on the complexity of CHAREs are summarized in Table 2. We denote by $\text{RE}(S)$ the set of all CHAREs. Recall that the three decision problems are PSPACE-complete for the class of all regular expressions [Koz77, SM73]. We briefly discuss our results:

- We show that inclusion is already coNP-complete for several, seemingly very innocent expressions: when every factor is of the form (i) $a$ or $a^*$, (ii) $a$ or $a^?$, (iii) $a$ or $(a_1^* + \cdots + a_n^*)$, (iv) $a$ or $w^+$ and (v) $a^+$ or $(a_1 + \cdots + a_n)$ with $a, a_1, \ldots, a_n$ arbitrary alphabet symbols and $w$ an arbitrary string with at least one symbol. Even worse, when factors of the form $(a_1 + \cdots + a_n)^*$ or $(a_1 + \cdots + a_n)^+$ are also allowed, we already obtain the maximum complexity: PSPACE. When such factors are disallowed the complexity remains coNP. The inclusion problem is in PTIME when we allow (general) regular expressions where the number of occurrences of the same symbol is bounded by some constant $k$ (a fragment we denote with $\text{RE}^{\leq k}$). As the running time is $n^k$, $k$ should of course be small to be feasible. Fortunately, this seems to be the case quite often. Recent investigation has pointed out that in practice, for ninety-nine percent of the regular expressions occurring in DTDs or XML Schemas, $k$ is equal to one [BNT].

- The precise complexity of equivalence largely remains open. Of course, it is never harder than inclusion, but we conjecture that it is tractable for a large fragment of $\text{RE}(S)$. We only prove a PTIME upper bound for expressions where each factor is $a$ or $a^*$, or $a$ or $a^?$. Even for these restricted fragments the proof is non-trivial. Basically, we show that two expressions are equivalent if and only if they have the same sequence normal form,
modulo one rewrite rule. Interestingly, the sequence normal form specifies factors much in the same way as XML Schema does. For every symbol, an explicit upper and lower bound is specified. For instance, \( aa^*bbc?c? \) becomes \( a[1, +]b[2, 2]c[0, 2] \).

- **intersection** is \( \text{coNP-complete} \) when each factor is either of the form (i) \( a \) or \( a^* \), (ii) \( a \) or \( a? \), (iii) \( a \) or \( (a_1^+ + \cdots + a_n^+) \), (iv) \( a \) or \( (a_1 + \cdots + a_n)^+ \) or of the form \( (v) a^+ \) or \( (a_1 + \cdots + a_n) \). As we can see, the complexity of intersection is not always the same as for inclusion. There are even cases where inclusion is harder and others where intersection is harder. In case (iv), for example, inclusion is \( \text{PSPACE-complete} \), whereas intersection problem is \( \text{coNP-complete} \). Indeed, intersection remains in \( \text{coNP} \) even if we allow all kinds of factors except \( (w_1 + \cdots + w_n)^+ \) or \( (w_1 + \cdots + w_n)^* \). On the other hand, intersection is \( \text{PSPACE-hard} \) for \( \text{RE} \leq 3 \) and for deterministic (or one-unambiguous) regular expressions [BKW98], whereas their inclusion problem is in \( \text{PTIME} \). The only tractable fragment we obtain is when each factor is restricted to \( a \) or \( a^+ \).

<table>
<thead>
<tr>
<th>RE-fragment</th>
<th>Inclusion</th>
<th>Equivalence</th>
<th>Intersection</th>
</tr>
</thead>
<tbody>
<tr>
<td>( a, a^+ )</td>
<td>\text{in PTIME (DFA)}</td>
<td>\text{in PTIME}</td>
<td>\text{in PTIME (6.6)}</td>
</tr>
<tr>
<td>( a, a^* )</td>
<td>\text{\text{coNP} (4.1)}</td>
<td>\text{in PTIME (5.1)}</td>
<td>\text{NP (6.1)}</td>
</tr>
<tr>
<td>( a, a? )</td>
<td>\text{\text{coNP} (4.1)}</td>
<td>\text{in PTIME (5.1)}</td>
<td>\text{NP (6.1)}</td>
</tr>
<tr>
<td>( a, (+a^+) )</td>
<td>\text{\text{coNP} (4.1)}</td>
<td>\text{in \text{coNP}}</td>
<td>\text{NP (6.1)}</td>
</tr>
<tr>
<td>( a^+, (+a) )</td>
<td>\text{\text{coNP} (4.1)}</td>
<td>\text{in \text{coNP}}</td>
<td>\text{NP (6.1)}</td>
</tr>
<tr>
<td>( a, w^+ )</td>
<td>\text{\text{coNP} (4.1)}</td>
<td>\text{in \text{coNP}}</td>
<td>\text{NP (6.1)}</td>
</tr>
<tr>
<td>( S - {(+, +w)^*, }(+, +w)^+ } )</td>
<td>\text{\text{coNP} (4.1)}</td>
<td>\text{in \text{coNP}}</td>
<td>\text{NP (6.1)}</td>
</tr>
<tr>
<td>( a, (+a)^* )</td>
<td>\text{\text{\text{PSPACE} (4.1)}}</td>
<td>\text{in \text{PSPACE}}</td>
<td>\text{NP (6.1)}</td>
</tr>
<tr>
<td>( a, (+a)^+ )</td>
<td>\text{\text{\text{PSPACE} (4.1)}}</td>
<td>\text{in \text{PSPACE}}</td>
<td>\text{NP (6.1)}</td>
</tr>
<tr>
<td>( S - {(+, +w)^*, }(+, +w)^+ } )</td>
<td>\text{\text{\text{PSPACE} (4.1)}}</td>
<td>\text{in \text{PSPACE}}</td>
<td>\text{\text{PSPACE ([Bal02])}}</td>
</tr>
<tr>
<td>( a, (+w)^* )</td>
<td>\text{\text{\text{PSPACE} (4.1)}}</td>
<td>\text{in \text{PSPACE}}</td>
<td>\text{\text{PSPACE ([Bal02])}}</td>
</tr>
<tr>
<td>( a, (+w)^+ )</td>
<td>\text{\text{\text{PSPACE} (4.1)}}</td>
<td>\text{in \text{PSPACE}}</td>
<td>\text{\text{PSPACE ([Bal02])}}</td>
</tr>
<tr>
<td>( S )</td>
<td>\text{\text{\text{PSPACE} (4.1)}}</td>
<td>\text{in \text{PSPACE}}</td>
<td>\text{\text{PSPACE ([Bal02])}}</td>
</tr>
<tr>
<td>( \text{RE} \leq k(k \geq 3) )</td>
<td>\text{\text{in PTIME (4.2)}}</td>
<td>\text{in PTIME}</td>
<td>\text{\text{PSPACE (6.4)}}</td>
</tr>
<tr>
<td>( \text{one-unambiguous} )</td>
<td>\text{\text{in PTIME}}</td>
<td>\text{in PTIME}</td>
<td>\text{\text{PSPACE (6.4)}}</td>
</tr>
</tbody>
</table>

Table 2: Summary of our results. Unless specified otherwise, all complexities are completeness results. The theorem numbers are given in brackets.

### 1.1 Related Work

The complexities of **equivalence**, **inclusion** and **intersection** for general regular expressions and several fragments were studied in [HRS76, Koz77, SM73]. From these, the most related result is the \( \text{coNP-completeness} \) of **equivalence** and **inclusion** of bounded languages [HRS76]. A language \( L \) is **bounded** if there
are strings \(v_1, \ldots, v_n\) such that \(L \subseteq v_1^* \cdots v_n^*\). It should be noted that the latter class is much more general than, for instance, our class \(\text{RE}(w^*)\). More recently, inclusion for two fragments of chain regular expressions have been shown to be tractable: inclusion for \(\text{RE}(a?, (+a)^*)\) [ABJ98] and \(\text{RE}(a, \Sigma, \Sigma^*)\) [MS99, MS04]. This last result should be contrasted with the \text{PSPACE}-completeness of inclusion for \(\text{RE}(a, (+a), (+a)^*)\), or even \(\text{RE}(a, (+a)^*)\). Further, Balas investigated intersection for regular expressions of limited star height [Bal02]. He showed that it is \text{PSPACE}-complete to decide whether \(\text{intersection}\) for \(\text{RE}((+w)^*)\) expressions contains a non-empty string. Balas’s proof can easily be adjusted to obtain \text{PSPACE}-hardness of \(\text{intersection}\) for \(\text{RE}(a, (+w)^*)\) or \(\text{RE}(a, (+w)^+\) expressions.

**Organization**  In Section 2, we introduce the necessary definitions and show some preliminary lemmas. In Section 3, we relate decision problems for XML schemas to the corresponding decision problems on regular expressions. In Section 4, 5, and 6, we consider inclusion, equivalence, and non-emptiness of intersection of chain regular expressions, respectively. In Section 7, we present concluding remarks.

## 2 Preliminaries

For the rest of the paper, \(\Sigma\) always denotes a finite alphabet. A \(\Sigma\)-symbol (or simply symbol) is an element of \(\Sigma\), and a \(\Sigma\)-string (or simply string) is a finite sequence \(w = a_1 \cdots a_n\) of \(\Sigma\)-symbols. We define the length of \(w\), denoted by \(|w|\), to be \(n\). We denote empty string by \(\epsilon\). The set of positions of \(w\) is \(\{1, \ldots, n\}\) and the symbol of \(w\) at position \(i\) is \(a_i\). By \(w_1 \cdot w_2\) we denote the concatenation of two strings \(w_1\) and \(w_2\). For readability, we sometimes also denote the concatenation of \(w_1\) and \(w_2\) by \(w_1 w_2\). The set of all strings is denoted by \(\Sigma^*\). A **string language** is a subset of \(\Sigma^*\). For two string languages \(L, L' \subseteq \Sigma^*\), we define their concatenation \(L \cdot L'\) to be the set \(\{w \cdot w' \mid w \in L, w' \in L'\}\). We abbreviate \(L \cdot L \cdots L \) (\(i\) times) by \(L^i\).

### 2.1 Regular expressions

The set of **regular expressions** over \(\Sigma\), denoted by \(\text{RE}\), is defined in the usual way:

- \(\emptyset\), \(\epsilon\), and every \(\Sigma\)-symbol is a regular expression; and
- when \(r\) and \(s\) are regular expressions, then \(rs\), \(r+s\), and \(r^*\) are also regular expressions.

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

- \(L(\emptyset) = \emptyset\);
• \( L(\varepsilon) = \{\varepsilon\} \);
• \( L(a) = \{a\} \);
• \( L(rs) = L(r) \cdot L(s) \);
• \( L(r + s) = L(r) \cup L(s) \);
• \( L(r^*) = \{\varepsilon\} \cup \bigcup_{i=1}^{\infty} L(r)^i \).

The size of a regular expression \( r \) over \( \Sigma \), denoted by \(|r|\), is the number of \( \Sigma \)-
symbols occurring in \( r \). By \( r^* \) and \( r^+ \), we abbreviate the expressions \( r + \varepsilon \) and \( rr^* \), respectively. Sometimes, we denote \( w \in L(r) \) simply by \( w \in r \).

In many of our proofs, we make use of how a string can be matched against a regular expression. We formalize this by the notion of a *match*. A *match* \( m \) between a string \( w = a_1 \cdots a_n \) and a regular expression \( r \) is a (partial) mapping from pairs \((i, j)\), \( 1 \leq i \leq j + 1 \leq n \), of positions of \( w \) to sets of subexpressions of \( r \). This mapping is consistent with the semantics of regular expressions, that is,

1. if \( \varepsilon \in m(i, j) \), then \( i = j + 1 \);
2. if \( a \in m(i, j) \), for \( a \in \Sigma \), then \( i = j \) and \( a_i = a \);
3. if \((r_1 + r_2) \in m(i, j)\), then \( r_1 \in m(i, j) \) or \( r_1 \in m(i, j) \);
4. if \( r_1 r_2 \in m(i, j) \), then there is a \( k \) such that \( r_1 \in m(i, k) \) and \( r_2 \in m(k+1, j) \);
5. if \( r^* \in m(i, j) \), then there are \( k_1, \ldots, k_t \) such that \( r \in m(i, k_1) \), \( r \in m(k_t + 1, j) \) and \( r \in m(k_{1}+1, k_{\ell+1}) \), for all \( \ell, 1 \leq \ell < t \).

Furthermore, \( m \) is minimal with these properties. That is, if \( m' \) fulfills (1)–(5) and \( m'(i, j) \subseteq m(i, j) \) for each \( i, j \), then \( m' = m \). We say that \( m \) matches a substring \( a_i \cdots a_j \) of \( w \) onto a subexpression \( r' \) of \( r \) when \( r' \in m(i, j) \). Often, we leave \( m \) implicit whenever this cannot give rise to confusion. We then say that \( a_i \cdots a_j \) matches \( r' \).

We consider simple regular expressions occurring in practice in DTDs and XML Schemas [BNV04], which we call CHAiN Regular Expressions (CHAREs). These regular expressions are defined as follows.

**Definition 2.1.** A *base symbol* is a regular expression \( s \), \( s^* \), \( s^+ \), or \( s? \), where \( s \) is a non-empty string; a *factor* is of the form \( e, e^+, e^* \), or \( e? \) where \( e \) is a disjunction of base symbols of the same kind. That is, \( e \) is of the form \((s_1 + \cdots + s_n), (s_1^* + \cdots + s_n^*), (s_1^+ + \cdots + s_n^+) \), or \((s_1? + \cdots + s_n?) \), 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 \(((abc)^* + b^*)(a + b)?(ab)^+(ac + b)^*\) is a chain regular expression. The expression \((a + b)^*(a^*b^*)\), however, is not.

7
We introduce 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 $\ast$, $+$, or $\cdot$. 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 $\ast$, $+$, or $\cdot$. A list of possible factors, together with their abbreviated notation, is displayed in Table 1. This table only shows factors which give rise to chain regular expressions with different expressive power.

We denote subclasses of chain regular expressions by $\text{RE}(X)$, where $X$ is a list of the allowed factors. For example, we write $\text{RE}((+a)^\ast, w?)$ for the set of regular expressions $e_1 \cdots e_n$ where every $e_i$ is either (i) $(a_1 + \cdots + a_m)^\ast$ 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 $(+a_1 + \cdots + a_n)^\ast$ simply by $A$. We denote the class of all chain regular expressions by $\text{RE}(S)$.

### 2.2 Decision problems

Two regular expressions $r, r'$ are included, denoted by $r \subseteq r'$, if $L(r) \subseteq L(r')$. They are equivalent, denoted by $r \equiv r'$, if $L(r) = L(r')$.

The following three problems are fundamental to this paper.

**Definition 2.2.** Let $\mathcal{R}$ be a class of regular expressions.

- **INCLUSION** for $\mathcal{R}$: Given two expressions $r, r' \in \mathcal{R}$, is $r \subseteq r'$?
- **EQUIVALENCE** for $\mathcal{R}$: Given two expressions $r, r' \in \mathcal{R}$, is $r \equiv r'$?
- **INTERSECTION** for $\mathcal{R}$: Given an arbitrary number of expressions $r_1, \ldots, r_n \in \mathcal{R}$, is $\bigcap_{i=1}^n L(r_i) \neq \emptyset$?

In the sequel, we abuse notation and denote $\bigcap_{i=1}^n L(r_i) \neq \emptyset$ simply by $\bigcap_{i=1}^n r_i \neq \emptyset$. Notice that for arbitrary regular expressions these problems are PSPACE-complete [Koz77, SM73].

### 2.3 Automata on compressed strings

We introduce some more notions that are frequently used in our proofs. We use the abbreviations NFA and DFA for non-deterministic and deterministic finite automata, respectively. Such automata are 5-tuples $(Q, \Sigma, \delta, I, F)$, where $Q$ is the state set, $\Sigma$ is the input alphabet, $\delta : Q \times \Sigma \to 2^Q$ is the transition function, $I$ is the set of initial states and $F$ is the set of final states. Furthermore, a DFA is an NFA with the property that $I$ is a singleton, and $\delta(q, a)$ is a singleton for every $q \in Q$ and $a \in \Sigma$.

A compressed string is a finite sequence of pairs $(w, i)$, where $w \in \Sigma^*$ is a string and $i > 0$ is a natural number. The pair $(w, i)$ stands for the string $w^i$. The size of a pair $(w, i)$ is $\lceil \log i \rceil$, plus the size of $w$. The size of a compressed
Finally, we accept when \( R \) is non-empty.

**Lemma 2.3.** Given a compressed string \( v \) and an NFA \( A \), we can test whether \( \text{string}(v) \in L(A) \) in time polynomial in the size of \( v \) and \( A \).

**Proof.** The idea is based on a proof by Meyer and Stockmeyer [SM73], which shows that the equivalence problem for regular expressions over a unary alphabet is \( \text{coNP} \)-complete. Let \( v = (w_1, i_1) \cdots (w_n, i_n) \) be a compressed string and let \( A = (Q_A, \Sigma, \delta_A, I_A, F_A) \) be an NFA with \( Q_A = \{1, \ldots, k\} \). We basically compute the set \( R \subseteq Q_A \) of states which are reachable from a state in \( I_A \) by reading \( \text{string}(v) \) from left to right. When we encounter a pair \((w, i)\), we compute \((M_{w})^{i}\), where \( M_w \) is the transition matrix of \( A \) on \( w \). The latter can be done by \( O(\log_2 i) \) matrix multiplications. Using \((M_{w})^{i}\), we then replace the current set \( R \) by the set of states which are reachable from a state in \( R \), by reading \( w^{i} \). If, in the end, \( R \cap F_A \) is non-empty, we accept.

We now describe this formally. We denote the canonical extension of \( \delta \) to strings in \( \Sigma^{*} \) by \( \hat{\delta} \). Initially, \( R = I_A \). We read \( v \) from left to right and repeatedly apply the following rule. When reading \((w_j, i_j)\), we distinguish two cases:

- \( i_j = 1 \): We replace \( \hat{\delta} \) with \( \hat{\delta}^{0}(\hat{\delta}, w_i), q \in R \). The latter can be done in time polynomial in the size of \( v \) and \( A \) in the straightforward manner.

- \( i_j > 1 \): Let \( M_w \) be the \( k \times k \) matrix such that for all \( \ell, m \in Q_A, M_w(\ell, m) = 1 \) if \( m \in \hat{\delta}(\ell, w) \) and \( M_w(\ell, m) = 0 \), otherwise. Then, we replace \( R \) by \( R' \) consisting of those \( q' \) for which \( q \in R \) and \( M_w^{i}(q, q') = 1 \). Notice that \( M_w \) can be computed in time polynomial in the size of \( v \) and \( A \). Furthermore, by applying the method of successive squaring [Sed83], \( M_w^{i} \) can be computed by \( O(\log_2 i) \) multiplications of \( k \times k \)-matrices.

Finally, we accept when \( R \cap F_A \) is non-empty. \( \square \)

### 2.4 Tiling systems

A **tiling system** is a tuple \( D = (T, H, V, \bar{b}, \bar{t}, n) \) where \( n \) is a natural number, \( T \) is a finite set of tiles; \( H, V \subseteq T \times T \) are horizontal and vertical constraints, respectively; and \( \bar{b}, \bar{t} \) are \( n \)-tuples of tiles (\( \bar{b} \) and \( \bar{t} \) stand for bottom row and top row, respectively). A **corridor tiling** is a mapping \( \lambda : \{1, \ldots, m\} \times \{1, \ldots, n\} \rightarrow T \), for some \( m \in \mathbb{N} \), such that \( \bar{b} = (\lambda(1, 1), \ldots, \lambda(1, n)) \) and \( \bar{t} = (\lambda(m, 1), \ldots, \lambda(m, n)) \). Intuitively, the first and last row of the tiling are \( \bar{b} \) and \( \bar{t} \), respectively. A tiling
is correct if it respects the horizontal and vertical constraints. That is, for every \( i = 1, \ldots, m \) and \( j = 1, \ldots, n - 1, (\lambda(i,j), \lambda(i,j+1)) \in H \), and for every \( i = 1, \ldots, m - 1 \) and \( j = 1, \ldots, n, (\lambda(i,j), \lambda(i+1,j)) \in V \).

To every tiling system we can associate a game as follows: the game consists of two players (CONSTRUCTOR and SPOILER). The game is played on an \( N \times n \) board. Each player places tiles in turn. While player CONSTRUCTOR tries to construct a corridor tiling, player SPOILER tries to prevent it. Player CONSTRUCTOR wins if SPOILER makes an illegal move (with respect to \( H \) or \( V \)), or when a correct corridor tiling can be constructed. We say that CONSTRUCTOR has a winning strategy if she wins no matter what SPOILER does.

In the sequel, we use reductions from the following problems:

- **CORRIDOR TILING**: given a tiling system, is there a correct corridor tiling?
- **TWO-PLAYER CORRIDOR TILING**: given a tiling system, does CONSTRUCTOR have a winning strategy?

The following theorem is due to Chlebus [Chl86].

**Theorem 2.4.** 1. **CORRIDOR TILING** is pspace-complete.

2. **TWO-PLAYER CORRIDOR TILING** is exptime-complete.

### 3 Decision Problems for DTDs and XML Schemas

As explained in the introduction, an important motivation for this study comes from reasoning about XML schemas. In this section, we describe how the basic decision problems for such schemas, namely whether two schemas describe the same set of documents or whether one describes a subset of the other, basically reduce to the equivalence and inclusion problem for regular expressions. We also address the problem whether a set of schemas define a common XML document. In the case of DTDs, the latter problem again reduces to the corresponding problem for regular expressions; for XML Schema Definitions (XSDs) it does not. The reader not interested in XML can safely skip this section.

#### 3.1 Trees

It is common to view XML documents as finite trees with labels from a finite alphabet \( \Sigma \). There is no limit on the number of children of a node. Figure 1 gives an example of an XML document, together with its tree representation. Of course, elements in XML documents can also contain references to nodes. But as XML schema languages usually do not constrain these, nor the data values at leaves, it is safe to view schemas as simply defining tree languages over a finite alphabet. In the rest of this section, we introduce the necessary background concerning XML schema languages.

The set of unranked \( \Sigma \)-trees, denoted by \( T_\Sigma \), is the smallest set of strings over \( \Sigma \) and the parenthesis symbols "(" and ")" such that, for \( \sigma \in \Sigma \) and \( w \in T_\Sigma \),...
Figure 1: An example of an XML document and its tree representations.

σ(w) is in $T_\Sigma$. So, a tree is either $\varepsilon$ (empty) or is of the form $\sigma(t_1 \cdots t_n)$ where each $t_i$ is a tree. In the tree $\sigma(t_1 \cdots t_n)$, the subtrees $t_1, \ldots, t_n$ are attached to the root labeled $\sigma$. We write $\sigma$ rather than $\sigma()$. Notice that there is no a priori bound on the number of children of a node in a $\Sigma$-tree; such trees are therefore unranked. For every $t \in T_\Sigma$, the set of nodes of $t$, denoted by Dom($t$), is the set defined as follows:

- if $t = \varepsilon$, then Dom($t$) = $\emptyset$; and
- if $t = \sigma(t_1 \cdots t_n)$, where each $t_i \in T_\Sigma$, then Dom = $\{\varepsilon\} \cup \bigcup_{i=1}^{n} \{iu \mid u \in$ Dom($t_i$)$\}$.

Figure 1(c) contains a tree in which we annotated the nodes between brackets.
Observe that the $n$ child nodes of a node $u$ are always $u_1, \ldots, u_n$, from left to right. The label of a node $u$ in the tree $t = \sigma(t_1 \cdots t_n)$, denoted by $\text{lab}^t(u)$, is defined as follows:

- if $u = \varepsilon$, then $\text{lab}^t(u) = \sigma$; and
- if $u = i u'$, then $\text{lab}^t(u) = \text{lab}^{t_i}(u')$.

We define the depth of a tree $t$, denoted by $\text{depth}(t)$, as follows: if $t = \varepsilon$, then $\text{depth}(t) = 0$; and if $t = \sigma(t_1 \cdots t_n)$, then $\text{depth}(t) = \max\{\text{depth}(t_i) \mid i = 1, \ldots, n\} + 1$. In the sequel, whenever we say tree, we always mean $\Sigma$-tree. A tree language is a set of trees.

### 3.2 XML Schema Languages

We use extended context-free grammars (ECFGs) and a restriction of extended DTDs [PV00] to abstract from DTDs and XML schemas.

**Definition 3.1.** Let $\mathcal{M}$ be a class of representations of regular string languages over $\Sigma$. A DTD is a tuple $(d, s_d)$ where $d$ is a function that maps $\Sigma$-symbols to elements of $\mathcal{M}$ 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 (i) $\text{lab}^t(\varepsilon) = s_d$ and, (ii) for every $u \in \text{Dom}(t)$ with $n$ children, $\text{lab}^t(u_1) \cdots \text{lab}^t(u_n) \in L(d(\text{lab}^t(u)))$. By $L(d)$ we denote the set of trees satisfying $d$.

For clarity, we write $a \rightarrow r$ rather than $d(a) = r$ in examples. In this notation, a simple example of a DTD defining the inventory of a store which sells DVDs is the following:

```
store  \rightarrow \ vdvd^* 
vdvd  \rightarrow \ title \ \ price 
title  \rightarrow \ \varepsilon 
price  \rightarrow \ \varepsilon.
```

The start symbol of the DTD is “store”. The DTD defines trees of depth two, where the root is labeled with “store” and has one or more children labeled with “vdvd”. Every “vdvd”-labeled node has two children labeled “title” and “price”, respectively.

We recall the definition of an extended DTD from [PV00]. The class of tree languages defined by EDTDs corresponds precisely to the regular (unranked) tree languages [BKMW01].

**Definition 3.2.** An extended DTD (EDTD) is a 4-tuple $E = (\Sigma, \Sigma', d, \mu)$, where $\Sigma'$ is an alphabet of types, $d$ is a DTD over $\Sigma'$, and $\mu$ is a mapping from $\Sigma'$ to $\Sigma$. Notice that $\mu$ 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 $L(E)$ the set of trees satisfying $E$.
In the sequel, we also denote by $\mu$ the homomorphic extension of $\mu$ to strings, trees, or regular expressions. For ease of exposition, we always take $\Sigma' = \{a^i \mid 1 \leq i \leq k, a \in \Sigma, i \in \mathbb{N}\}$ for some natural numbers $k$, and we set $\mu(a^i) = a$.

For a node $u$ in a $\Sigma$-tree $t$, we say that $a^i$ is a type of $u$ with respect to $E$ when

1. there exists a $\Sigma'$-tree $t' \in L(d)$ such that $\mu(t') = t$; and
2. $\text{lab}^t(u) = a^i$.

For convenience of notation, we usually leave the EDTD $E$ implicit whenever this cannot give rise to confusion.

For simplicity, we also denote EDTDs in examples in a similar way as DTDs, that is, we write $a^i \rightarrow r$ rather than $d(a^i) = r$. A simple example of an EDTD is the following:

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>store$^1$</td>
<td>$\rightarrow$</td>
<td>(dvd$^1 + dvd^2)^* dvd^2 (dvd^1 + dvd^2)^* dvd^2 (dvd^1 + dvd^2)^*$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>dvd$^1$</td>
<td>$\rightarrow$</td>
<td>title$^1$ price$^1$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>dvd$^2$</td>
<td>$\rightarrow$</td>
<td>title$^1$ price$^1$ discount$^1$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>title$^1$</td>
<td>$\rightarrow$</td>
<td>$\varepsilon$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>price$^1$</td>
<td>$\rightarrow$</td>
<td>$\varepsilon$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>discount$^1$</td>
<td>$\rightarrow$</td>
<td>$\varepsilon$</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Here, dvd$^1$ defines ordinary DVDs, while dvd$^2$ defines DVDs on sale. The rule for store$^1$ specifies that there should be at least two DVDs on sale.

The following restriction on EDTDs corresponds to the expressiveness of XML Schema [MLMK05].

**Definition 3.3.** A **single-type EDTD** (EDTD$^{st}$) is an EDTD $(\Sigma, \Sigma', d, \mu)$ with the property that for every $a \in \Sigma'$, in the regular expression $d(a)$ no two types $b^i$ and $b^j$ with $i \neq j$ occur.

The above defined EDTD is not single-type as both dvd$^1$ and dvd$^2$ occur in the rule for store$^1$. A simple example of a single-type EDTD is the following:

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>store$^1$</td>
<td>$\rightarrow$</td>
<td>regulars$^1$ discounts$^1$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>regulars$^1$</td>
<td>$\rightarrow$</td>
<td>$(dvd^1)^*$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>discounts$^1$</td>
<td>$\rightarrow$</td>
<td>dvd$^2$ dvd$^2$ (dvd$^2$)$^*$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>dvd$^1$</td>
<td>$\rightarrow$</td>
<td>title$^1$ price$^1$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>dvd$^2$</td>
<td>$\rightarrow$</td>
<td>title$^1$ price$^1$ discount$^1$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>title$^1$</td>
<td>$\rightarrow$</td>
<td>$\varepsilon$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>price$^1$</td>
<td>$\rightarrow$</td>
<td>$\varepsilon$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>discount$^1$</td>
<td>$\rightarrow$</td>
<td>$\varepsilon$</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Although there are still two element definitions dvd$^1$ and dvd$^2$, they can only occur in different right hand sides.

The following restriction of EDTDs corresponds to the semantic notion of **one-pass preorder typing**: it characterizes precisely the tree languages that allow
to type documents in a streaming fashion. That is, when traversing a tree in a depth-first left-to-right order, they allow to determine the type of every node when it is met for the first time. Or, more formally, the type that can be assigned to a node \( v \) in a tree \( t \) is functionally determined by the preceding of \( v \) in \( t \) and the label of \( v \) itself [MNS05, BMNS05]. We illustrate this in Figure 2. Here, the preceding of \( v \) is the shaded area in \( t \).

**Definition 3.4.** Let \( E = (\Sigma, \Sigma', d, \mu) \) be an EDTD. A regular expression \( r \) over \( \Sigma' \) restrains competition if there are no strings \( w a^i v \) and \( w a^j v' \) in \( L(r) \) with \( i \neq j \). An EDTD \( E \) is restrained competition (EDTDrc) if all regular expressions occurring in the definition of \( d \) restrain competition.

An example of a restrained competition EDTD that is not single-type is given next:

\[
\begin{align*}
\text{store}^1 & \rightarrow (\text{dvd}^1)^* \text{discounts}^1 (\text{dvd}^2)^* \\
\text{discounts}^1 & \rightarrow \varepsilon \\
\text{dvd}^1 & \rightarrow \text{title}^1 \text{price}^1 \\
\text{dvd}^2 & \rightarrow \text{title}^1 \text{price}^1 \text{discount}^1 \\
\text{title}^1 & \rightarrow \varepsilon \\
\text{price}^1 & \rightarrow \varepsilon \\
\text{discount}^1 & \rightarrow \varepsilon 
\end{align*}
\]

Notice that any single-type EDTD is also restrained competition. The classes of tree languages defined by the grammars introduced above are included as follows: \( \text{DTD} \subset \text{EDTD}^\text{at} \subset \text{EDTD}^\text{rc} \subset \text{EDTD} \) [MLMK05].

**Remark 3.5.** Murata et al. observed that there is a very simple deterministic algorithm to check validity of a tree \( t \) with respect to a EDTDat or EDTDrc \( E \) [MLMK05, MNS05]. It proceeds top-down and assigns to every node with some symbol \( a \) a type \( a' \). To the root the start symbol of \( d \) is assigned; then, for every interior node \( u \) with type \( a' \), it is checked whether the children of \( u \) match \( \mu(d(a')) \); if not, the tree is rejected; otherwise, as \( E \)'s restrained competition, to each child a unique type can be assigned. The tree is accepted, if this process terminates at the leaves without any rejection.

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 \( u \in \text{Dom}(t) \) such that \( \text{lab}^t(u) = a \).
Hence, for example, the DTD \((d, a)\) where \(d(a) = a\) is not reduced. We can reduce a DTD in four steps:

1. Compute in a bottom-up manner the set of all symbols \(S = \{\sigma \mid L((d, \sigma)) \neq \emptyset\}\).
2. Replace every occurrence of a symbol in \(\Sigma \setminus S\) in a regular expression by \(\emptyset\), and remove all symbols in \(\Sigma \setminus S\) from the DTD.
3. In the resulting DTD, compute the set \(R \subseteq \Sigma\) of symbols that are reachable from the start symbol \(s\) of the extended context free grammar.
4. Replace every occurrence of a symbol in \((\Sigma \setminus S) \setminus R\) in a regular expression by \(\emptyset\), and remove all symbols in \((\Sigma \setminus S) \setminus R\) from the DTD.

Step (1) is known to be in \(\text{ptime}\), as shown in [MN05].\(^3\) Step (3) can be carried out in \(\text{ptime}\) by the straightforward reachability algorithm. An EDTD is reduced if its DTD is reduced. Unless mentioned otherwise, we assume in the sequel that all DTDs and EDTDs are reduced.

We consider the same decision problems for XML schemas as for regular expressions.

**Definition 3.6.** Let \(\mathcal{M}\) be a subclass of the class of DTDs, EDTDs, EDTD\(\text{st}\)s or EDTD\(\text{rc}\)s.

- **inclusion for \(\mathcal{M}\):** Given two schemas \(d, d' \in \mathcal{M}\), is \(L(d) \subseteq L(d')\)?
- **equivalence for \(\mathcal{M}\):** Given two schemas \(d, d' \in \mathcal{M}\), is \(L(d) = L(d')\)?
- **intersection for \(\mathcal{M}\):** Given an arbitrary number of schemas \(d_1, \ldots, d_n \in \mathcal{M}\), is \(\bigcap_{i=1}^{n} L(d_i) \neq \emptyset\)?

### 3.3 Inclusion and Equivalence of XML Schema Languages

As already mentioned, testing equivalence and inclusion of XML schema languages is related to testing equivalence and inclusion of regular expressions. It is immediate that complexity lower bounds for regular expressions imply lower bounds for XML schema languages. A consequence is that testing equivalence and inclusion of XML schemas is \(\text{pspace}\)-hard, which suggests looking for simpler regular expressions.

Interestingly, in the case of the practically important DTDs and single-type EDTDs, it turns out that the complexities of the equivalence and inclusion problem on strings also imply **upper bounds** for the corresponding problems on XML trees.

\(^3\)Actually, it was shown that testing emptiness of nondeterministic unranked tree automata with NFAs as representation for the internal regular languages, is in \(\text{ptime}\). As converting a DTD into such a tree automaton is straightforward (and in \(\text{ptime}\)), the result is immediate.
For a class \( R \) of regular expressions, we denote by DTD(\( R \)), EDTD(\( R \)), EDTD\(^{st} \)(\( R \)), and EDTD\(^{rc} \)(\( R \)), the class of DTDs, EDTDs, EDTD\(^{st} \) and EDTD\(^{rc} \)s with regular expressions in \( R \).

We call a complexity class \( C \) closed under positive reductions if the following holds for every \( O \in C \). Let \( L' \) be accepted by a deterministic polynomial-time Turing machine \( M \) with oracle \( O \) (denoted \( L' = L(M^O) \)). Let \( M \) further have the property that \( L(M^A) \subseteq L(M^B) \) whenever \( A \subseteq B \). Then \( L' \) is also in \( C \). For a more precise definition of this notion we refer the reader to [HO02]. For our purposes, it is sufficient that important complexity classes like PTIME, NP, CONP, and PSPACE have this property, and that every such class contains PTIME.

Let, for an alphabet symbol \( a \), \( S_a \) be the set \( \{a\} \cup \{a^i \mid i \in \mathbb{N}\} \). We say that a homomorphism \( h \) on strings is label-preserving if \( h(S_a) \subseteq S_a \) for every \( \Sigma \)-symbol \( a \). We call a class of regular expressions \( R \) closed under label-preserving homomorphisms if, for every \( r \in R \) and every label-preserving homomorphism \( h, h(r) \in R \).

We now show in Theorem 3.7 that deciding inclusion or equivalence for DTDs, EDTD\(^{st} \)s, or EDTD\(^{rc} \)s is essentially not harder than deciding inclusion or equivalence for the regular expressions that they use. Notice that a theorem similar to this one appeared as Theorem 15 in [MNS05], but the present theorem is more general.

**Theorem 3.7.** Let \( R \) be a class of regular expressions which is closed under label-preserving homomorphisms and \( C \) be a complexity class which is closed under positive reductions. Then the following are equivalent:

(a) inclusion for \( \mu(R) \) expressions is in \( C \).

(b) inclusion for DTD(\( \mu(R) \)) is in \( C \).

(c) inclusion for EDTD\(^{st} \)(\( R \)) is in \( C \).

(d) inclusion for EDTD\(^{rc} \)(\( R \)) is in \( C \).

The corresponding statements holds for equivalence.

**Proof.** The implications from (d) to (c) and (b) to (a) are immediate.

For the implication from (c) to (b), let \( d_1 \) and \( d_2 \) be two DTD(\( R \))s. Let, for every \( i = 1, 2, d'_i \) be the DTD obtained from \( d_i \) by replacing every regular expression \( d_i(a) \) by \( \nu(d_i(a)) \), where \( \nu \) is the label-preserving homomorphism mapping every \( \Sigma \)-symbol \( b \) to the type \( b^1 \). As \( R \) is closed under label-preserving homomorphisms, we have that the EDTD\(^{st} \)s \( E_1 = (\Sigma, \Sigma', d'_1, \mu) \) and \( E_2 = (\Sigma, \Sigma', d'_2, \mu) \) are in EDTD\(^{st} \)(\( R \)). It is easy to see that \( L(d_1) = L(E_1) \) and \( L(d_2) = L(E_2) \). Hence, \( L(d_1) \subseteq L(d_2) \) (respectively, \( L(d_1) = L(d_2) \)) if and only if \( L(E_1) \subseteq L(E_2) \) (respectively, \( L(E_1) = L(E_2) \)).

It remains to prove that (a) implies (d). To this end, let \( E_1 = (\Sigma, \Sigma'_1, (d_1, s_{d_1}), \mu_1) \) and \( E_2 = (\Sigma, \Sigma'_2, (d_2, s_{d_2}), \mu_2) \) be two reduced EDTD\(^{rc} \)(\( R \))s. We define a correspondence relation \( R \subseteq \Sigma'_1 \times \Sigma'_2 \) as follows:

1. \((s_{d_1}, s_{d_2}) \in R\); and,
(2) if \((a^i, a^j) \in R\), \(w_1 b^k v_1 \in L(d_1(a^i))\), \(w_2 b^\ell v_2 \in L(d_2(a^j))\) and \(\mu_1(w_1) = \mu_2(w_2)\) then \((b^k, b^\ell) \in R\).

We need the following observation further in the proof. The observation follows immediately from the restrained competition property of \(E_1\) and \(E_2\) and the fact that \(E_1\) and \(E_2\) are reduced.

**Observation 3.8.** A pair \((a^i, a^j)\) is in \(R\) if and only if there is a tree \(t\) with a node \(u\) labeled \(a\), such that:

1. the algorithm of Remark 3.5 assigns type \(a^i\) to \(u\) with respect to \(E_1\); and
2. the algorithm of Remark 3.5 assigns type \(a^j\) to \(u\) with respect to \(E_2\).

Notice that we do not require that \(t\) matches \(E_1\) or \(E_2\) overall.

We show that the relation \(R\) can be computed in \(\text{PTIME}\) in a top-down left-to-right manner. To this end, let \(A_{a^i} = (Q_{a^i}, \Sigma_1, \delta_{a^i}, I_{a^i}, F_{a^i})\) and \(A_{a^j} = (Q_{a^j}, \Sigma_2, \delta_{a^j}, I_{a^j}, F_{a^j})\) be the Glushkov-automata of \(d_1(a^i)\) and \(d_2(a^j)\), respectively (see [Glu61, BKW98]). We now give an \(\text{NLOGSPACE}\) decision procedure that tests, given \((a^i, a^j) \in R\), whether there are \(w_1 b^k v_1 \in L(d_1(a^i))\) and \(w_2 b^\ell v_2 \in L(d_2(a^j))\) with \(\mu_1(w_1) = \mu_2(w_2)\). The algorithm guesses the strings \(w_1 b^k\) and \(w_2 b^\ell\) one symbol at a time, while simulating \(A_{a^i}\) on \(w_1 b^k\) and \(A_{a^j}\) on \(w_2 b^\ell\). For both strings, we only remember the last symbol we guessed. We also only remember one state per automaton. Initially, this is the start state of \(A_{a^i}\) and the start state of \(A_{a^j}\). Every time, after guessing a symbol \(x_1\) of \(w_1 b^k\) and a symbol \(x_2\) of \(w_2 b^\ell\), we verify whether \(\mu_1(x_1) = \mu_2(x_2)\) and we overwrite the current states \(q_1, x_1\) and \(q_2, x_2\), respectively. We nondeterministically determine when we stop guessing the strings \(w_1 b^k\) and \(w_2 b^\ell\). The algorithm is successful when, from the moment that we stopped guessing, final states of \(A_{a^i}\) and \(A_{a^j}\) are reachable from the current states \(q_1, x_1\) and \(q_2, x_2\), which can also be decided in \(\text{NLOGSPACE}\). As we only remember two states and two alphabet symbols at the same time, we only use logarithmic space.

We now show how testing inclusion and equivalence for \(\text{EDTDFA}\) reduces to testing inclusion and equivalence for the regular expressions. This follows from Claim 3.9 below and the closure property of \(\mathcal{C}\). The statement for equivalence follows likewise.

**Claim 3.9.** With the notation as above, \(L(E_1)\) is included in (equivalent with) \(L(E_2)\) if and only if, for every \(a^i \in \Sigma_1\) and \(a^j \in \Sigma_2\) with \((a^i, a^j) \in R\), the regular expression \(\mu_1(d_1(a^i))\) is included in (equivalent with) \(\mu_2(d_2(a^j))\).

**Proof.** We give a proof for inclusion. Equivalence then immediately follows. We can assume without loss of generality that \(\mu_1(s) = \mu_2(s)\).

\((\Rightarrow)\) We prove this direction by contraposition. Suppose that there is a pair \((a^i, a^j) \in R\) for which the regular expression \(\mu_1(d_1(a^i))\) is not included in \(\mu_2(d_2(a^j))\). We then need to show that \(L(E_1)\) is not included in \(L(E_2)\).
Thereto, let \( w \) be a counterexample \( \Sigma \)-string in \( L(\mu_1(d_1(a^i))) - L(\mu_2(d_2(a^j))) \).

From Observation 3.8, we now know that there exists a tree \( t \in L(E_1) \), with a node \( u \in \text{Dom}(t) \), such that the following holds:

- the type assigned to \( u \) with respect to \( E_1 \) is \( a^i \);
- the type assigned to \( u \) with respect to \( E_2 \) is \( a^j \); and
- the concatenation of the labels of \( u \)'s children is \( w \).

Obviously, \( t \) is not in \( L(E_2) \) as \( u \)'s children do not match \( \mu_2(d_2(a^j)) \). So, \( L(E_1) \) is not included in \( L(E_2) \).

\((\Leftarrow)\) We prove this direction by contraposition. Suppose that \( L(E_1) \) is not included in \( L(E_2) \), so there is a tree \( t \) matching \( E_1 \) but not \( E_2 \). We need to show that there is an \( a^i \in \Sigma_1' \) and \( a^j \in \Sigma_2' \), with \((a^i, a^j) \in R\), such that \( \mu_1(d_1(a^i)) \) is not included in \( \mu_2(d_2(a^j)) \).

As \( t \not\in L(E_2) \), there is a node to which the algorithm in Remark 3.5 assigns a type \( a^j \) of \( E_2 \), but for which the concatenation of the labels of the children do not match the regular expression \( \mu_2(d_2(a^j)) \). Let \( u \in \text{Dom}(t) \) be such a node such that no other node on the path in \( t \) from the root to \( u \) has this property. Let \( a^i \) be the type that the algorithm in Remark 3.5 assigns to \( u \) with respect to \( E_1 \). So, by Observation 3.8, we have that \((a^i, a^j) \in R\). Let \( w \) be the concatenation of the labels of \( u \)'s children. But as \( w \in L(\mu_1(d_1(a^i))) \), and \( w \) is not in \( L(\mu_2(d_2(a^j))) \), we have that \( \mu_1(d_1(a^i)) \) is not included in \( \mu_2(d_2(a^j)) \).

This concludes the proof of Theorem 3.7

An interesting corollary of this theorem is that inclusion is in \( \text{Ptime} \) for \( \text{EDTD}^{rc}(\mathcal{R}) \) where \( \mathcal{R} \) is the class of deterministic restrained competition expressions.

### 3.4 Intersection of DTDs and XML Schemas

We show in this section that the complexity of the intersection problem for regular expressions is an upper bound for the corresponding problem on DTDs. In Theorem 3.11, we show how the intersection problem for DTDs reduces to testing intersection of the regular expressions that are used in the DTDs.

Unfortunately, a similar property probably does not hold for the case for single-type EDTDs or restrained competition EDTDs, as we show in Theorem 3.12. Theorem 3.12 shows that there is a class of EDTDs for which the intersection problem is \( \text{EXPTIME} \)-hard. As the intersection problem for regular expressions is \( \text{PSPACE} \)-complete, the intersection problem for single-type or restrained competition EDTDs cannot be reduced to the corresponding problem for regular expressions, unless \( \text{EXPTIME} = \text{PSPACE} \).

We start by showing that the intersection problem for DTDs can be reduced to the corresponding problem for regular expressions. Thereto, let \( \mathcal{R} \) be a class of regular expressions. The \textit{generalized intersection} problem for \( \mathcal{R} \) is to
determine, given an arbitrary number of expressions \( r_1, \ldots, r_n \in \mathcal{R} \) and a set \( S \subseteq \Sigma \), whether \( \bigcap_{i=1}^{n} r_i \cap S^* \neq \emptyset \).

We first show that the intersection problem and the generalized intersection problem are equally complex.

**Lemma 3.10.** For a \( \mathcal{R} \) a class of regular expressions and \( \mathcal{C} \) a complexity class closed under positive reductions. The following are equivalent:

(a) The intersection problem for \( \mathcal{R} \) is in \( \mathcal{C} \).

(b) The generalized intersection problem for \( \mathcal{R} \) is in \( \mathcal{C} \).

**Proof.** We only prove the direction from (a) to (b). Let \( r_1, \ldots, r_n \) be regular expressions in \( \mathcal{R} \) and let \( S \subseteq \Sigma \). Let \( r'_i \) be obtained from \( r_i \) by replacing every occurrence of a symbol in \( \Sigma - S \) by the regular expression \( \emptyset \). Then, \( \bigcap_{i=1}^{n} r'_i \neq \emptyset \) if and only if \( \bigcap_{i=1}^{n} r_i \cap S^* \neq \emptyset \). \( \square \)

We are now ready to show the theorem for DTDs.

**Theorem 3.11.** Let \( \mathcal{R} \) be a class of regular expressions and let \( \mathcal{C} \) be a complexity class which is closed under positive reductions. Then the following are equivalent:

(a) The intersection problem for \( \mathcal{R} \) expressions is in \( \mathcal{C} \).

(b) The intersection problem for DTD(\( \mathcal{R} \)) is in \( \mathcal{C} \).

**Proof.** We only prove the direction from (a) to (b), as the reverse direction is trivial. Thereto, let \( d_1, \ldots, d_n \) be in DTD(\( \mathcal{R} \)). We assume without loss of generality that \( d_1, \ldots, d_n \) all have the same start symbol. We compute the set of symbols \( S_i \) with the following property:

\[
a \in S_i \text{ if and only if there is a tree of depth at most } i \\
\text{with root labeled } a \text{ in } L((d_1, a)) \cap \cdots \cap L((d_n, a)).
\]

\((\ast)\)

Initially, we set \( S_1 = \{ a \mid \varepsilon \in L(d_j(a)) \text{ for all } j \in \{1, \ldots, n\} \} \). For every \( i > 1 \), \( S_i \) is the set \( S_{i-1} \) extended with all symbols \( a \) for which \( S_{i-1}^* \cap d_i(a) \cap \cdots \cap d_n(a) \neq \emptyset \). Clearly, \( S_{k+1} = S_k \) for \( |\Sigma| = k \). It can be shown by a straightforward induction on \( i \) that \((\ast)\) holds. So, there is a tree satisfying all DTDs if and only if the start symbol belongs to \( S_k \).

It remains to argue that \( S_k \) can be computed in \( \mathcal{C} \). Clearly, for any set of regular expressions, it can be checked in PTIME whether their intersection accepts \( \varepsilon \). So, \( S_1 \) can be computed in PTIME and hence in \( \mathcal{C} \). From Lemma 3.10, it follows that \( S_i \) can be computed from \( S_{i-1} \) in \( \mathcal{C} \). As only \( k \) sets need to be computed, the overall algorithm is in \( \mathcal{C} \). This concludes the proof of Theorem 3.11. \( \square \)
The following theorem shows that single-type and restrained competition EDTDs probably cannot be included in Theorem 3.11. Indeed, by Theorem 6.1, the intersection problem for $\text{RE}((+a), w?, (+a)?, (+a)^*)$ expressions is in NP and by Theorem 3.12, the intersection problem is already EXPTIME-complete for single-type EDTDs with $\text{RE}((+a), w?, (+a)?, (+a)^*)$ expressions. The proof of Theorem 3.12 is similar to the proof that intersection of deterministic top-down tree automata is EXPTIME-complete [Sei94]. However, single-type EDTDs and the latter automata are incomparable. Indeed, the tree language consisting of the trees \{a(bc), a(cb)\} is not definable by a top-down deterministic tree automaton, while it is by the EDTD consisting of the rules $a^1 \rightarrow b^1 c^1 + c^1 b^1$, $b^1 \rightarrow \varepsilon$, $c^1 \rightarrow \varepsilon$. Conversely, the tree language \{a(b(c)b(d))\} is not definable by a single-type EDTD, but is definable by a top-down deterministic tree automaton.

**Theorem 3.12.** The intersection problem for $\text{EDTD}^{st}((+a), w?, (+a)?, (+a)^*)$ is EXPTIME-hard.

**Proof.** We use a reduction from TWO-PLAYER CORRIDOR TILING. Let $D = (T, H, V, \bar{b}, \bar{t}, n)$ be a tiling system with $T = \{t_1, \ldots, t_k\}$. We construct several single-type EDTDs such that their intersection is non-empty if and only if player CONSTRUCTOR has a winning strategy.

As $\Sigma$ we take $T \cup \{\#\}$. We define $d_0$ to be a single-type EDTD defining all possible strategy trees. Every path in such a tree will encode a tiling. The root is labeled with $\#$. Inner nodes are labeled with tiles. Nodes occurring on an even depth are placed by player CONSTRUCTOR and have either no children or have every tile in $T$ as a child representing the choice of SPOILER. Nodes occurring on an odd depth are placed by player SPOILER and have either no children or precisely one child representing the choice of CONSTRUCTOR. The start symbol of $d_0$ is $\#$. The EDTD $d_0$ uses the alphabet $\Sigma' = \{\#, \text{error}, t_1^1, \ldots, t_k^1, t_1^2, \ldots, t_k^2\}$. The rules are as follows:

- $\# \rightarrow (t_1^1 + \cdots + t_k^1)$;
- for every $t \in T$, $t^1 \rightarrow (t_1^2 \cdots t_k^2)^?; \text{ and}$
- for every $t \in T$, $t^2 \rightarrow (t_1^1 + \cdots + t_k^1 + \text{error})^?$.

Here, a tile with label $t_i$ is assigned the type $t_i^1$ (respectively $t_i^2$) if it corresponds to a move of player CONSTRUCTOR (respectively SPOILER). We use the special symbol error to mark that player SPOILER has placed a wrong tile. Notice that $\bar{b}$ and $\bar{t}$ are not present in the tree.

All other single-type EDTDs will check the correct shape of the tree and the horizontal and vertical constraints.

First, we make the following observation. Let $M = (Q, \Sigma, \delta, \{0\}, F)$ be a DFA with state set $Q = \{0, \ldots, m\}$, with the property that there is an $a \in \Sigma$ such that every string in $L(M)$ starts with $a$. Then, a single-type EDTD $d_M$ can be constructed in LOGSPACE defining the trees over $\Sigma$ for which every path from the root to a leaf is accepted by $M$. Indeed, we let $a^0$ be the start symbol.

20
Then, for every $i, j \in Q$ and $b \in \Sigma$ for which $\delta(i, b) = \{j\}$, $d_M$ contains the rule $b^i \rightarrow (t_1^i + \cdots + t_k^i + \#^i)^*$ if $i \in F$, and $b^i \rightarrow (t_1^i + \cdots + t_k^i + \#^i)^+$, otherwise.

Let $M_1, \ldots, M_\ell$ be a sequence of DFAs that check the following properties:

- Every string starts with $\#$, has no other occurrences of $\#$, and either (i) ends with error or (ii) the length of every string is one modulo $n$. This can be checked by one automaton.
- All horizontal constraints are satisfied, or player SPOILER places the first tile which violates the horizontal constraints. This tile is followed by the special symbol error. This can be checked by one automaton.
- For every position $i = 1, \ldots, n$, all vertical constraints on tiles on a position $i \pmod n$ are satisfied, or player SPOILER places a the first tile which violates the vertical constraint. This tile is followed by the special symbol error. This can be checked by $n$ automata.
- For every position $i = 1, \ldots, n$, the $i$th tile of $b$ and the $i$th tile of the first row should satisfy the vertical constraints. This can be checked by one automaton.
- For every position $i = 1, \ldots, n$, the $i$th tile of the last row and the $i$th tile of $t$ should satisfy the vertical constraints. This can be checked by one automaton.

Clearly, $d_0 \cap d_{M_1} \cap \cdots \cap d_{M_\ell}$ is non-empty if and only if player CONSTRUCTOR has a winning strategy. 

### 4 Inclusion

We now turn to the complexity of INCLUSION, EQUIVALENCE, and INTERSECTION for the chain regular expressions themselves. We start our investigation with the inclusion problem. As mentioned before, it is PSPACE-complete for general regular expressions. The following tractable cases have been identified in the literature:

- **INCLUSION** for $RE(a?, (+a)^*)$ can be solved in linear time. Whether $p \subseteq p_1 + \cdots + p_k$ for $p, p_1, \ldots, p_k$ from $RE(a?, (+a)^*)$ can be checked in quadratic time [ABJ98].
- In [MS99] it is stated that INCLUSION for $RE(a, \Sigma, \Sigma^*)$ is in PTIME. A proof can be found in [MS04].

Some of the fragments we defined are so small that one expects their containment problem to be tractable. Therefore, it comes as a surprise that even for $RE(a, a?)$ and $RE(a, a^*)$ the inclusion problem is already coNP-complete. Even worse, for $RE(a, (+a)^*)$ and $RE(a, (+a)^+)$ we already obtain the maximum complexity: PSPACE-completeness. The PSPACE-hardness of inclusion of
RE(a, (+a)*) expressions should be contrasted with the PTIME inclusion for RE(a, Σ, Σ*) obtained in [MS99], where disjunctions can only range over the complete alphabet.

Our results, together with corresponding upper bounds are summarized in Theorem 4.1, which we prove in a series of lemmas (Lemma a–c). Let RE(S − {(+a)*, (+w)*, (+a)+, (+w)+}) denote the fragment of RE(S) where no factors of the form (a1 + · · · + an)*, (w1 + · · · + wn)*, (a1 + · · · + an)+ or (w1 + · · · + wn)+ are allowed for n ≥ 2.

Theorem 4.1. (a) Inclusion is coNP-hard for
   (1) RE(a, a*)
   (2) RE(a, a?)
   (3) RE(a, (+a+))
   (4) RE(a, w+)
   (5) RE(a+, (+a))

(b) Inclusion is PSPACE-hard for
   (1) RE(a, ( +a)+); and
   (2) RE(a, (+a)*)

(c) Inclusion is in coNP for RE(S − {(+a)*, (+w)*, (+a)+, (+w)+}); and

(d) Inclusion is in PSPACE for RE(S).

Theorem 4.1 does not leave much room for tractable cases. Of course, inclusion is in PTIME for any class of regular expressions for which expressions can be transformed into DFAs in polynomial time. An easy example of such a class is RE(a, a+).

Another example has probably more importance in practice. Often, the same symbol occurs only a few times in a regular expression of a DTD. As a matter of fact, if we impose a fixed bound k on the number of such occurrences, then the containment problem becomes tractable. For every k, let RE≤k denote the class of all regular expressions where every symbol can occur at most k times.

Theorem 4.2. Inclusion for RE≤k is in PTIME.

Proof. Let r be an RE≤k expression. Let Ar = (Q, Σ, δ, I, F) be the Glushkov automaton for r [Glu61] (see also [BKW98]). As the states of this automaton are basically the positions in r, after reading a symbol there are always at most k possible states in which the automaton might be. Therefore, determinizing A only leads to a DFA of size |A|^k. As k is fixed, inclusion of such automata is in PTIME.
It should be noted though that the upper bound for the running time is $O(n^k)$, therefore $k$ should be very small to be really useful. Fortunately, this seems to be the case in many practical scenarios. Indeed, recent investigation has pointed out that in practice, for ninety-nine percent of the regular expressions DTDs or XML Schemas, $k$ is equal to one [BNT].

In the rest of this section, we prove Theorem 4.1.

**Proof of Theorem 4.1(a)** We show that for all five cases, there is a logspace reduction from validity of propositional 3DNF formulas. The validity problem asks, given a propositional formula $\Phi$ in 3DNF with variables $\{x_1, \ldots, x_n\}$, whether $\Phi$ is true under all truth assignments for $\{x_1, \ldots, x_n\}$. The validity problem for 3DNF formulas is known to be $\text{coNP}$-complete [GJ79]. We note that, for the cases (1–3), we even show that inclusion is already $\text{coNP}$-hard when the expressions use a fixed-size alphabet.

Our proof technique is inspired by a proof of Miklau and Suciu, showing that the inclusion problem for XPath expressions with predicates, wildcard, and the axes “child” and “descendant” is $\text{coNP}$-hard [MS04]. We present a robust generalization and use it to show $\text{coNP}$-hardness of all five fragments.

We now proceed with the proof. Thereto, let $\Phi = C_1 \lor \cdots \lor C_k$ be a propositional formula in 3DNF using variables $\{x_1, \ldots, x_n\}$. In the five cases, we construct regular expressions $R_1, R_2$ such that

$$L(R_1) \subseteq L(R_2) \text{ if and only if } \Phi \text{ is valid.}$$

More specifically, we encode truth assignments for $\Phi$ by strings. The basic idea is to construct $R_1$ and $R_2$ such that $L(R_1)$ contains all string representations of truth assignments and a string $w$ matches $R_2$ if and only if $w$ represents an assignment which makes $\Phi$ true.

We discuss the building blocks of expressions $R_1$ and $R_2$. Let $U$ be a regular expression describing exactly one string $u$. In this proof, $U$ is either of the form $a^n$ or of the form $\#a^i\# \cdots \#a^i\#$ ($n$ occurrences of $a^i$ separated by $\#$) for some integer $i$. We construct a regular expression $W$ such that the strings of $L(W)$ can be interpreted as truth assignments. More precisely, for each truth assignment $A$, there is a string $w_A \in L(W)$ and for each string $w \in L(W)$ there is a corresponding truth assignment $A_w$. Then we set

$$R_1 = U^kWU^k,$$
$$R_2 = NF_1 \cdots F_kN,$$

where $N$ and $F_i, i = 1, \ldots, k$, are regular expressions for which the following properties hold:

(i) $u^i \in L(N)$ for every $i = 1, \ldots, k$.

(ii) If $A_w$ makes $C_i$ true then $w \in L(F_i)$. If $w_A \in L(F_i)$ then $A$ makes $C_i$ true.

(iii) $u \in L(F_i)$ for every $i = 1, \ldots, k$.

(iv) If $u^kwu^k \in L(R_1) \cap L(R_2)$, then $w$ matches some $F_i$. 

23
We first show the following claim:

**Claim 4.3.** If there are expressions \( U, W, N, F_1, \ldots, F_k \) satisfying (i)–(iv) above, then
\[
L(R_1) \subseteq L(R_2) \text{ if and only if } \Phi \text{ is valid.}
\]

**Proof.** Suppose that there are expressions \( U, W, N, F_1, \ldots, F_k \) satisfying (i)–(iv). We prove that \( L(R_1) \subseteq L(R_2) \) if and only if \( \Phi \) is valid.

\((\Rightarrow)\) Assume that \( L(R_1) \subseteq L(R_2) \). Let \( A \) be an arbitrary truth assignment for \( \Phi \) and let \( v = u^k w_A u^k \). As \( v \in L(R_1) \) and \( L(R_1) \subseteq L(R_2) \), we also have that \( v \in L(R_2) \). By (iv), it follows that \( w_A \) matches some \( F_i \). Hence, by (ii), \( C_i \) is made true by \( A \). Consequently, \( \Phi \) is also made true by \( A \). As \( A \) is an arbitrary truth assignment, we have that \( \Phi \) is a valid propositional formula.

\((\Leftarrow)\) Suppose that \( \Phi \) is valid. Let \( v \) be an arbitrary string in \( L(R_1) \). By definition of \( R_1 \), \( v \) is of the form \( v = u^k w u^k \), where \( u \) is the unique string in \( L(U) \). Consider the truth assignment \( A_w \) corresponding to \( w \). As \( \Phi \) is valid, there is a clause \( C_i \) of \( \Phi \) that becomes true under \( A_w \). Due to (ii), we have that \( w \in L(F_i) \). Furthermore, as \( u \) matches \( F_j \) for every \( j = 1, \ldots, k \) (by (iii)), and as \( L(N) \) contains \( u^\ell \) for every \( \ell = 1, \ldots, k \) (by (i)), we have that \( v = u^k w u^k \in L(R_2) \). As \( v \) is an arbitrary string in \( L(R_1) \), we have that \( L(R_1) \subseteq L(R_2) \). This completes the proof of the claim. \( \square \)

It remains to construct regular expressions \( U, W, N, F_1, \ldots, F_k \) with the required properties. In all five cases, we construct these expressions starting from five basic regular expressions \( r_{\text{true}}, r_{\text{false}}, r_{\text{true,false}}, r_{\text{all}}, \) and \( \alpha \), which must adhere to the inclusion structure graphically represented in Figure 3 and formally defined by the properties (INC1)–(INC5) below. Intuitively, the two dots in Figure 3 are strings \( z_{\text{true}} \) and \( z_{\text{false}} \), which represent the truth values \text{true} and \text{false}, respectively. The expressions \( r_{\text{true}} \) and \( r_{\text{false}} \) are used to match \( z_{\text{true}} \) and \( z_{\text{false}} \) in \( R_2 \), respectively. The expression \( r_{\text{true,false}} \) is used to generate all truth assignments which must then be matched in \( R_2 \). Finally, \( \alpha \) and \( r_{\text{all}} \) are used to ensure that condition (iv) above holds. That is, they ensure that when \( L(R_1) \subseteq L(R_2) \), every string generated by \( W \) must match some \( F_i \).

In all cases, the expressions \( r_{\text{true}}, r_{\text{false}}, r_{\text{true,false}}, r_{\text{all}}, \) and \( \alpha \) have the properties (INC1)–(INC5):

\[
\begin{align*}
\alpha & \in L(r_{\text{false}}) \cap L(r_{\text{true}}) \quad \text{(INC1)} \\
L(r_{\text{true,false}}) & \subseteq L(r_{\text{false}}) \cup L(r_{\text{true}}) \quad \text{(INC2)} \\
L(r_{\text{true,false}}) \cup \{\alpha\} & \subseteq L(r_{\text{all}}) \quad \text{(INC3)} \\
z_{\text{true}} & \in L(r_{\text{true,false}}) - L(r_{\text{false}}) \quad \text{(INC4)} \\
z_{\text{false}} & \in L(r_{\text{true,false}}) - L(r_{\text{true}}) \quad \text{(INC5)}
\end{align*}
\]

For the first three fragments, we now define the needed expressions. We deal with the other two fragments later. Note that the alphabet size is fixed (at most four) in the reduction for these fragments.
Figure 3: Inclusion structure of regular expressions used in coNP-hardness of inclusion.

(1) For RE(a, a*):
- $\alpha = a$;
- $r_{true} = a a^* b^* a^*$;
- $r_{false} = b^* a^*$;
- $r_{true,false} = r_{all} = a^* b^* a^*$;
- $z_{true} = ab$;
- $z_{false} = ba$;
- $U = \#a\alpha\# \cdots \#a\# \ (n \ occurrences \ of \ \alpha)$;
- $W = \#r_{true,false}\# \cdots \#r_{true,false}\# \ (n \ occurrences \ of \ r_{true,false})$; and
- $N = (\#a^* a^* b^* a^* \cdots a^* a^* #)^k \ (n \ occurrences \ of \ a^* \ in \ each \ of \ the \ k \ copies)$.

(2) For RE(a, a?):
- $\alpha = a$;
- $r_{true} = a a?$;
- $r_{false} = a?$;
- $r_{true,false} = r_{all} = a? a?$;
- $z_{true} = aa$;
- $z_{false} = \varepsilon$;
- $U = \#a\alpha\# \cdots \#a\# \ (n \ occurrences \ of \ \alpha)$;
- $W = \#r_{true,false}\# \cdots \#r_{true,false}\# \ (n \ occurrences \ of \ r_{true,false})$; and
- $N = (\#a^* a^* a^* \cdots a^* a^* #)^k \ (n \ occurrences \ of \ a^* \ in \ each \ of \ the \ k \ copies)$.

(3) For RE(a, w+):
• \( \alpha = aaaa; \)
• \( r_{\text{true}} = a^+(aa)^+; \)
• \( r_{\text{false}} = (aa)^+; \)
• \( r_{\text{true, false}} = aa^+; \)
• \( r_{\text{all}} = a^+; \)
• \( z_{\text{true}} = aaaa; \)
• \( z_{\text{false}} = aa; \)
• \( U = \# \alpha \# \alpha \# \cdots \# \alpha \# (n \text{ occurrences of } \alpha); \)
• \( W = \# r_{\text{true, false}} \# \cdots \# r_{\text{true, false}} \# (n \text{ occurrences of } r_{\text{true, false}}); \) and
• \( N = (\# aaaa\$aaaa\$ \cdots \$aaaa\#)^+ (n \text{ occurrences of } aaaa) \)

It is straightforward to verify that the conditions (INC1)–(INC5) are fulfilled for each of the fragments.

With \( w = \# w_1 \# \cdots \# w_n \# \in L(W) \) we associate a truth assignment \( A_w \) as follows:

\[
A_w(x_j) := \begin{cases} 
\text{true}, & \text{if } w_j \in L(r_{\text{true}}); \\
\text{false}, & \text{otherwise.}
\end{cases}
\]

Let \( z_{\text{false}} \in L(r_{\text{true, false}}) - L(r_{\text{true}}) \) and \( z_{\text{true}} \in L(r_{\text{true, false}}) - L(r_{\text{false}}). \) They exist by conditions (INC4) and (INC5). For a truth assignment \( A, \) let

\[ w_A = \# w_1 \# \cdots \# w_n \# , \]

where, for each \( j = 1, \ldots, n, \) \( w_j = z_{\text{true}} \) if \( A(x_j) = \text{true} \) and \( w_j = z_{\text{false}}, \) otherwise.

For each \( i = 1, \ldots, k, \) we set

\[ F_i = \# e_1 \# \cdots \# e_n \# , \]

where for each \( j = 1, \ldots, n, \)

\[
e_j := \begin{cases} 
\text{false}, & \text{if } x_j \text{ occurs negated in } C_i, \\
\text{true}, & \text{if } x_j \text{ does not occur negated in } C_i, \\
\text{all}, & \text{otherwise.}
\end{cases}
\]

It remains to show that, for each of the fragments, conditions (i)–(iv) hold:

(i) Trivial.

(ii) Let \( w = \# w_1 \# \cdots \# w_n \# \in W \) be a string for which \( A_w \) makes \( C_i \) true and let \( F_i = \# e_1 \# \cdots \# e_n \# \) be as defined above. We need to show that \( w \in L(F_i). \) Thereeto, let \( j \leq n \) be an arbitrary positive integer. We need to consider three cases:
1. If $x_j$ does not occur in $C_i$ then $e_j = r^{\text{all}}$, by definition of $e_j$. Hence, as $w_j \in L(r^{\text{true, false}})$, and by condition (INC3), we have that $w_j \in L(e_j)$.

2. If $x_j$ occurs positively, then $e_j = r^{\text{true}}$, by definition of $e_j$. As $A_w$ makes $C_i$ true, we know that $A_w(x_j) = \text{true}$. By definition of $A_w$, we know that $w_j \in L(r^{\text{true}}) = L(e_j)$.

3. If $x_j$ occurs negatively, then $e_j = r^{\text{false}}$, by definition of $e_j$. As $A_w$ makes $C_i$ true, we know that $A_w(x_j) = \text{false}$. As $w_j \in r^{\text{true, false}}$ and $w_j \notin L(r^{\text{true}})$ by definition of $A_w(x_j)$, condition (INC2) gives that $w_j \in L(r^{\text{false}}) = L(e_j)$.

As $j \leq n$ is an arbitrarily chosen positive integer, we have that for each $j = 1, \ldots, n, w_j \in L(e_j)$. Consequently, we also have that $w \in L(F_i)$.

We show the other statement by contraposition. Thereto, let $A$ be a truth assignment such that $C_i$ is a clause not fulfilled by $A$. We need to show that $w_A \notin L(F_i)$. We need to consider two cases:

1. Suppose there exists an $x_j$ which occurs positively in $C_i$ and $A(x_j)$ is false. By definition of $F_i$, the $e_j$ component of $F_i$ is $r^{\text{true}}$ and, by definition of $w_A$, the $w_j$ component of $w_A$ is $z^{\text{false}} \notin L(r^{\text{true}})$. Hence, $w_A \notin L(F_i)$.

2. Otherwise, there exists an $x_j$ which occurs negatively in $C_i$ and $A(x_j)$ is true. By definition of $F_i$, the $e_j$ component of $F_i$ is $r^{\text{false}}$ and, by definition of $w_A$, the $w_j$ component of $w_A$ is $z^{\text{true}} \notin L(r^{\text{false}})$. Hence, $w_A \notin L(F_i)$.

(iii) This follows immediately from conditions (INC1), (INC3), and the definition of $F_i$.

(iv) Suppose that $u^k w u^k \in L(R_1) \cap L(R_2)$. We need to show that $w$ matches some $F_i$. Observe that the strings $u, w$, and every string in every $L(F_i)$ is of the form $\#y\#$ where $y$ is a non-empty string over the alphabet $\{a, b, \}$.

Also, every string in $L(N)$ is of the form $\#y_1\#\#y_2\#\cdots\#y_l\#$, where $y_1, \ldots, y_l$ are non-empty strings over $\{a, \}$. Hence, as $u^k w u^k \in L(R_2)$ and as none of the strings $y, y_1, \ldots, y_l$ contain the symbol “$\#$”, we have that $w$ either matches some $F_i$ or $w$ matches a sub-expression of $N$.

We now distinguish between fragments (1–2) and fragment (3). Let $m$ be a match between $u^k w u^k$ and $R_2$.

- In fragments (1–2) we have that $\ell \leq k$. Towards a contradiction, assume that $m$ matches a superstring of $u^k w$ to the left occurrence of $N$ in $R_2$. Note that $u^\ell w$ is a string of the form $\#y_1\#\#y_2\#\cdots\#y_{k+1}\#$, where $y_1, \ldots, y_{k+1}$ are non-empty strings over $\{a, b, \}$. But as $\ell \leq k$, no superstring of $u^k w$ can match $N$, which is a contradiction. Analogously, no superstring of $wu^k$ matches the right occurrence of the expression $N$ in $R_2$. So, $m$ must match $w$ onto some $F_i$. 

27
In fragment (3), we have that $\ell \geq 1$. Again, towards a contradiction, assume that $m$ matches a superstring of $u^k w$ onto the left occurrence of the expression $N$ in $R_2$. Observe that every string that matches $F_1 \cdots F_k N$ is of the form $\#y_1'' \# \# y_2'' \# \cdots \# y'_n \#$, where $\ell' > k$. As $u^k$ is not of this form, $m$ cannot match $u^k$ onto $F_1 \cdots F_k N$, which is a contradiction. Analogously, $m$ cannot match a superstring of $wu^k$ onto the right occurrence of the expression $N$ in $R_2$. So, $m$ must match $w$ onto some $F_i$.

This proves Lemma a for fragments (1)–(3).

We still need to deal with the fragments (4) and (5). The main difference with the fragments (1)–(3) is that we will no longer use an alphabet with fixed size. Instead, we use the symbols $b_j$ and $c_j$, for $j = 1, \ldots, n$. Instead of the basic regular expressions $r_{\text{true}}$, $r_{\text{false}}$, $r_{\text{true,false}}$, and $r_{\text{all}}$, we will now have expressions $r_{j,\text{true}}$, $r_{j,\text{false}}$, $r_{j,\text{true,false}}$, and $r_{j,\text{all}}$ for every $j = 1, \ldots, n$. We will require that these expressions have the same properties (INC1)–(INC5), but only between the expressions $r_{j,\text{true}}$, $r_{j,\text{false}}, r_{j,\text{true,false}}$, and $r_{j,\text{all}}$ with the same index $j$, and $\alpha$.

The needed regular expressions are then defined as follows:

(4) For $\text{RE}(a, (+a^+))$:

- $\alpha = a$;
- $r_{j,\text{true}} = (a^+ + b_j^+)$;
- $r_{j,\text{false}} = (a^+ + c_j^+)$;
- $r_{j,\text{true,false}} = (b_j^+ + c_j^+)$;
- $r_{j,\text{all}} = (a^+ + b_j^+ + c_j^+)$;
- $z_{j,\text{true}} = b_j$;
- $z_{j,\text{false}} = c_j$;
- $U = \alpha^n$;
- $W = r_{1,\text{true,false}} \cdots r_{n,\text{true,false}}$;
- $N = a^+$

(5) For $\text{RE}(a^+, (+a))$:

- $\alpha = a$;
- $r_{j,\text{true}} = (a + b_j)$;
- $r_{j,\text{false}} = (a + c_j)$;
- $r_{j,\text{true,false}} = (b_j + c_j)$;
- $r_{j,\text{all}} = (a + b_j + c_j)$;
- $z_{j,\text{true}} = b_j$;
- $z_{j,\text{false}} = c_j$;
\[ U = \alpha^n; \]
\[ W = r_1^{\text{true,false}} \cdots r_n^{\text{true,false}}; \] and
\[ N = a^+ \]

With \( w = w_1 \cdots w_n \in L(W) \), where for every \( j = 1, \ldots, n \), \( w_j \in r_j^{\text{true,false}} \), we associate a truth assignment \( A_w \) as follows:
\[
A_w(x_j) := \begin{cases} 
\text{true}, & \text{if } w_j \in L(r_j^{\text{true}}); \\
\text{false}, & \text{otherwise.}
\end{cases}
\]

Let \( z_j^{\text{false}} \in L(r_j^{\text{true,false}}) - L(r_j^{\text{true}}) \) and \( z_j^{\text{true}} \in L(r_j^{\text{true,false}}) - L(r_j^{\text{false}}) \). They exist by conditions (INC4) and (INC5). For a truth assignment \( A \), let
\[
w_A = w_1 \cdots w_n,
\]
where, for each \( j = 1, \ldots, n \), \( w_j = z_j^{\text{true}} \) if \( A(x_j) = \text{true} \) and \( w_j = z_j^{\text{false}} \), otherwise.

For each \( i = 1, \ldots, k \), we set
\[
F_i = e_1 \cdots e_n,
\]
where for each \( j = 1, \ldots, n \),
\[
e_j := \begin{cases} 
\text{false}, & \text{if } x_j \text{ occurs negated in } C_i; \\
\text{true}, & \text{if } x_j \text{ does not occur negated in } C_i; \\
\text{all}, & \text{otherwise.}
\end{cases}
\]

We show that, for fragments (4) and (5), conditions (i)–(iv) hold:

(i) Trivial.

(ii) This can be shown analogously as for the fragments (1)–(3).

(iii) This follows immediately from conditions (INC1), (INC3), and the definition of \( F_i \).

(iv) Suppose that \( u^k w u^k \in L(R_1) \cap L(R_2) \). We need to show that \( w \) matches some \( F_i \). To this end, let \( m \) be a match between \( u^k w u^k \) and \( R_2 \). For every \( j = 1, \ldots, n \), let \( \Sigma_j \) denote the set \( \{b_j, c_j\} \). Observe that the string \( w \) is of the form \( y_1 \cdots y_n \), where, for every \( j = 1, \ldots, n \), \( y_j \) is a string in \( \Sigma_j^+ \). Moreover, no strings in \( L(N) \) contain symbols from \( \Sigma_j \) for any \( j = 1, \ldots, n \). Hence, \( m \) cannot match any symbol of the string \( w \) onto \( N \). Consequently, \( m \) matches the entire string \( w \) onto a subexpression of \( F_1 \cdots F_k \) in \( R_2 \).

Further, observe that every string in every \( F_i \), \( i = 1, \ldots, k \), is of the form \( y'_1 \cdots y'_n \), where each \( y'_j \) is a string in \( (\Sigma_j \cup \{a\})^+ \). As \( m \) can only match symbols in \( \Sigma_j \) onto subexpressions with symbols in \( \Sigma_j \), \( m \) matches \( w \) onto some \( F_i \).
This concludes the proof of Lemma a. □

We prove Theorem 4.1(b) by a reduction from CORRIDOR TILING.

**Proof of Theorem 4.1(b)** We first show that INCLUSION is PSPACE-hard for RE($a, (+a)^+$) and we consider the case of RE($a, (+a)^*$) later. In both cases, we use a reduction from the CORRIDOR TILING problem, which is known to be PSPACE-complete.

To this end, let $D = (T, H, V, \bar{b}, \bar{t}, n)$ be a tiling system. Without loss of generality, we assume that $n \geq 2$. We construct two regular expressions $R_1$ and $R_2$ such that

$$R_1 \subseteq R_2$$

if and only if there exists no correct corridor tiling for $D$.

Let $\Sigma_i = \{t_i \mid t \in T\}$, which is the alphabet we will use to tile the $i$-th column.

Set $\Sigma = \bigcup_{i=1}^{n} \Sigma_i$. For ease of exposition, we denote $\Sigma \cup \{$ by $\Sigma_\#$, and $\Sigma \cup \#$ by $\Sigma_\#,\#$. We encode candidates for a correct tiling by a string in which the rows are separated by the symbol $\#$, that is, by strings of the form

$$\bar{b} \# \Sigma_1^+ \cdots \Sigma_n^+ \bar{t} \#.$$

The following regular expressions detect strings of this form which do not encode a correct tiling:

- $\Sigma_1^+ t_i t'_j \Sigma_1^+$, for every $t, t' \in T$, where $i = 1, \ldots, n - 1$ and $j \neq i + 1$. These expressions detect consecutive symbols that are not from consecutive column sets;

- $\Sigma_1^+ \bar{b} \Sigma_1^+$ for every $i \neq 1$ and $t_i \in \Sigma_i$, and $\Sigma_1^+ t_i \Sigma_1^+$ for every $i \neq n$ and $t_i \in \Sigma_i$. These expressions detect rows that do not start or end with a correct symbol. Together with the previous expressions, these expressions detect all candidates with at least one row not in $\Sigma_1 \cdots \Sigma_n$.

- $\Sigma_1^+ t_i t'_{i+1} \Sigma_1^+$, for every $(t, t') \not\in H$, and $i = 1, \ldots, n - 1$. These expressions detect all violations of horizontal constraints.

- $\Sigma_1^+ t_i \Sigma_1^+ t'_i \Sigma_1^+$, for every $(t, t') \not\in \Sigma$ and for every $i = 1, \ldots, n$. These expressions detect all violations of vertical constraints.

Let $e_1, \ldots, e_k$ be an enumeration of the above expressions. Notice that $k = O(|D|^4)$. It is straightforward that a string $w$ in (†) does not match $\bigcup_{i=1}^{k} e_i$ if and only if $w$ encodes a correct tiling.

Let $e = e_1 \cdots e_k$. Because of leading and trailing $\Sigma_1^+$ expressions, $L(e) \subseteq L(e_i)$, for every $i = 1, \ldots, k$. We are now ready to define $R_1$ and $R_2$:

$$R_1 = \underbrace{\# e \# e \# \cdots \# e \#}_{k \text{ times } e} \bar{b} \# \Sigma_1^+ \# \Sigma_1^+ \# \bar{t} \# \Sigma_1^+ \# e \# e \# \cdots \# e \#; \text{ and,}$$

$$R_2 = \underbrace{\Sigma_\# \# e_1 \# e_2 \# \cdots \# e_k \#}_{k \text{ times } e} \Sigma_\# \#. $$

30
Notice that both $R_1$ and $R_2$ are in $\text{RE}(a, (+a)^+)$ and can be constructed in polynomial time. It remains to show that $R_1 \subseteq R_2$ if and only if there is no correct tiling for $D$.

We first show the implication from left to right. Thereto, let $R_1 \subseteq R_2$. Let $uuw'$ be an arbitrary string in $L(R_1)$ such that $u, u' \in L(\#e\#e\# \cdots \#e\#)$ and $w \in \#\#e^+\#\#$. Hence, $uuw' \in L(R_2)$. Let $m$ be a match between $uuw'$ and $R_2$. Notice that $uuw'$ contains $2k + 2$ times the symbol "#". However, as $uuw'$ starts and ends with the symbol "#", $m$ matches the first and the last "#" of $uuw'$ onto the $\Sigma^+_{e, \#}$ sub-expressions of $R_2$ (1). This means that $m$ matches $k + 1$ consecutive #-symbols of the remaining $2k$ #-symbols in $uuw'$ onto the #-symbols in $#e_1#e_2# \cdots #e_k#$. Hence, $m$ matches $w$ onto some $e_i$. So, $w$ does not encode a correct tiling. As the sub-expression $\#\#e^+\#\#$ of $R_1$ defines all candidate tilings, the system $D$ has no solution.

To show the implication from right to left, assume that there is a string $uuw' \in L(R_1)$ that is not in $R_2$, where $u, u' \in L(\#e\#e\# \cdots \#e\#)$. Then $w \notin \bigcup_{i=1}^{k} L(e_i)$ and, hence, $w$ encodes a correct tiling.

The PSPACE-hardness proof for $\text{RE}(a, (+a)^+)$ is completely analogous, except that every "+" (which is not a disjunction) has to be replaced by a "\*" and that $R_2 = #\Sigma^+_{e, \#} #e_1#e_2# \cdots #e_k# \Sigma^+_{e, \#}$. The addition of the start and end symbol "#" is to enforce the condition (1) above.

**Proof of Theorem 4.1(c)** Let $r_1, r_2$ be expressions in $\text{RE}(S-\{(+a)^*, (+w)^*, (+a)^+, (+w)^+\})$. By translating the regular expressions to NFAs and deterministic them, it is easy to see that when $r_1 \not\subseteq r_2$, there is a counterexample string $s$ of at most exponential size in $|r_1| + |r_2|$, such that $s \in L(r_1)$ but $s \notin L(r_2)$. Let $N_{r_1, r_2}$ denote this size.

Because of the restricted form of $r_1$, it is possible to encode $s$ as a compressed string $s'$ of size polynomial in $|r_1| + |r_2|$. Indeed, $r_1$ is of the form $e_1 \cdots e_n$ where each $e_i$ is of the form $(w_1 + \cdots + w_k)$, $(w_1 + \cdots + w_k)^+$, $(w_1^+ + \cdots + w_k^+)$ or $(w_1^+ + \cdots + w_k^+)$, for $k \geq 1$ and $w_1, \ldots, w_k \in \Sigma^+$. Hence, $s$ can be written as $s_1 \cdots s_n$, where each $s_i$ matches $e_i$. Unless $e_i$ is of the form $(w_1^+ + \cdots + w_k^+)$ or $(w_1^+ + \cdots + w_k^+)$, $s_i$ is of small size, that is, smaller than or equal to $|r_1|$. However, if $e_i$ is of the form $(w_1^+ + \cdots + w_k^+)$ or $(w_1^+ + \cdots + w_k^+)$, then $s_i = w_j^+$, for some $j, \ell$ where $\ell \leq N_{r_1, r_2}$. So, the binary representation of $\ell$ has polynomial length in $|r_1| + |r_2|$. We therefore represent $s_i$ by the pair $(w_j, \ell)$. In all other cases, $|s_i| \leq |e_i|$ and we represent $s_i$ simply by $(s_i, 1)$. So, it suffices to guess for every $e_i$ a pair $(w_j, \ell)$ where $w_i$ occurs in $e_i$ and $\ell \leq N_{r_1, r_2}$. According to Lemma 2.3, we can verify in polynomial time in the size of the compressed string $s'$ that $s' \notin L(r_2)$. This concludes the proof of (c).

Notice that, in the proof of Lemma c, we actually did not make use of the restricted structure of the expression $r_2$. We can therefore state the following corollary:

**Corollary 4.4.** Let $r_1$ be an RE$(S-\{(+a)^*, (+w)^*, (+a)^+, (+w)^+\})$ expression and let $r_2$ be an arbitrary regular expression. Then, deciding whether $L(r_1) \subseteq L(r_2)$ is in connp.
Theorem 4.1(d) holds as the containment problem for arbitrary regular expressions is in PSPACE. This concludes the proof of Theorem 4.1.

5 Equivalence

In the present section, we merely initiate the research on the equivalence of chain regular expressions. Of course, upper bounds for INCLUSION imply upper bounds for EQUIVALENCE, but testing equivalence can be simpler. We show that the problem is in PTIME for $\text{RE}(a, a?)$ and $\text{RE}(a, a^*, a^*)$ by showing that such expressions are equivalent if and only if they have a corresponding sequence normal form (defined below). We conjecture that EQUIVALENCE remains tractable for larger fragments, or even the full fragment of chain regular expressions. However, showing that EQUIVALENCE is in PTIME is already non-trivial for $\text{RE}(a, a?)$ and $\text{RE}(a, a^*, a^*)$ expressions.

We now define the required normal form for $\text{RE}(a, a?)$ and $\text{RE}(a, a^*, a^*)$ expressions. To this end, let $r = r_1 \cdots r_n$ be a chain regular expression with factors $r_1, \ldots, r_n$. The sequence normal form of $r$ is obtained in the following way. First, we replace every factor of the form

- $s$ by $s[1, 1]$;
- $s?$ by $s[0, 1]$;
- $s^*$ by $s[0, *]$;
- $s^+$ by $s[1, *],$

where $s$ is an alphabet symbol. We call $s$ the base symbol of the factor $s[i, j]$. Then, we replace successive subexpressions $s[i_1, j_1]$ and $s[i_2, j_2]$ with the same base symbol $s$ by

- $s[i_1 + i_2, j_2 + j_2]$ when $j_1$ and $j_2$ are integers; and by
- $s[i_1 + i_2, *]$ when $j_1 = *$ or $j_2 = *$,

until no such replacements can be made anymore. For instance, the sequence normal form of $aa?a+a?bb?b^*$ is $a[2, 4]b[1, *]$. When $r'$ is the sequence normal form of a chain regular expression, and $s[i, j]$ is a subexpression of $r'$, then we call $c[i, j]$ a factor of $r'$.

Unfortunately, there are equivalent $\text{RE}(a, a^*)$ expressions that do not share the same sequence normal form. For instance, the regular expressions

$$r_1(a, b) = a[i, *]b[0, *]a[0, *]b[l, *]a[l, *]$$

and

$$r_2(a, b) = a[i, *]b[l, *]a[l, *]$$

are equivalent but have different sequence normal forms. So, whenever an expression of the form $r_1(a, b)$ occurs, it can be replaced by $r_2(a, b)$. The strong
sequence normal form of an expression \( r \) is the expression \( r' \) obtained by applying this rule as often as possible. It is easy to see that \( r' \) is unique.

We extend the notion of a match between a string and a regular expression in the obvious way to expressions in (strong) sequence normal form.

**Theorem 5.1.** EQUIVALENCE is in PTIME if for

1. \( RE(a,a?) \), and
2. \( RE(a,a^*,a^+) \).

*Proof.* We prove that, for both fragments, two expressions are equivalent only if they have the same strong sequence normal form.

We introduce some notions. If \( f \) is an expression of the form \( e[i,j] \), we write \( \text{base}(f) \) for \( e \), \( \text{upp}(f) \) for the upper bound \( j \) and \( \text{low}(f) \) for the lower bound \( i \). If \( r = r_1 \cdots r_n \) is an expression in sequence normal form, we write \( \text{max}(r) \) for the maximum upper bound in \( r \) different from \( * \), that is, for \( \max\{\text{upp}(r_i) \mid \text{upp}(r_i) \neq * \} \). 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 \( i = 1, \ldots, n-1 \), \( a_i \neq a_{i+1} \) and \( v \) is of the form \( a_i^{k_i} \) for some \( i \).

In the following, we prove that if two expressions \( r \) and \( s \) from one of the two stated fragments are equivalent, their strong sequence normal forms \( r' \) and \( s' \) are equal. The proof is a case study which eliminates one by one all differences between the strong normal forms of the two equivalent expressions.

Therefore, let \( r \) and \( s \) be two equivalent expressions and let \( r' = r_1 \cdots r_n \) and \( s' = s_1 \cdots s_m \) be the strong sequence normal form of \( r \) and \( s \), respectively. We assume that every \( r_1, \ldots, r_n \) and \( s_1, \ldots, s_m \) is of the form \( e[i,j] \). Let \( k := 1 + \max(\text{max}(r'), \text{max}(s')) \), that is, \( k \) is larger than any upper bound in \( r' \) and \( s' \) different from \( * \).

We first show that \( m = n \) and that, for every \( i = 1, \ldots, n \), \( \text{base}(r_i) = \text{base}(s_i) \). Thereto, let \( v^{\text{max}} = v_1^{\text{max}} \cdots v_n^{\text{max}} \), where, for every \( i = 1, \ldots, n \),

\[
  v_i^{\text{max}} = \begin{cases} 
  \text{base}(r_i)^k & \text{if upp}(r_i) = *, \\
  \text{base}(r_i)^{\text{upp}(r_i)} & \text{otherwise}.
\end{cases}
\]

Obviously, \( v^{\text{max}} \) is an element of \( L(r) \). Hence, by our assumption that \( r \) is equivalent to \( s \), we also have that \( v^{\text{max}} \in L(s) \). As \( v^{\text{max}} \) contains \( n \) blocks, \( s' \) must have at least \( n \) factors, so \( m \geq n \). Correspondingly, we define the string \( u^{\text{max}} = u_1^{\text{max}} \cdots u_m^{\text{max}} \), where, for every \( j = 1, \ldots, m \),

\[
  u_j^{\text{max}} = \begin{cases} 
  \text{base}(s_j)^k & \text{if upp}(s_j) = *, \\
  \text{base}(s_j)^{\text{upp}(s_j)} & \text{otherwise}.
\end{cases}
\]

As \( u^{\text{max}} \) has to match \( r \), we can conclude that \( r' \) has at least \( m \) factors, so \( n \geq m \). Hence, we obtain that \( m = n \). Furthermore, for each \( i = 1, \ldots, n \), it follows immediately that \( \text{base}(r_i) = \text{base}(s_i) \).

We now show that, for every \( i = 1, \ldots, n \), \( \text{upp}(r_i) = \text{upp}(s_i) \). If, for some \( i \), \( \text{upp}(r_i) = * \) then \( v_i^{\text{max}} \) has to be matched in \( s' \) by a factor with upper bound \( * \).
The analogous statement holds if \( \text{upp}(s_i) = * \). Hence, \( \text{upp}(r_i) = * \) if and only if \( \text{upp}(s_i) = * \). Finally, we similarly get that \( \text{upp}(r_i) = \text{upp}(s_i) \) for factors \( r_i, s_i \) with \( \text{upp}(r_i) \neq * \) and \( \text{upp}(s_i) \neq * \).

It only remains to show for each \( i = 1, \ldots, n \), that we also have that \( \text{low}(r_i) = \text{low}(s_i) \). By considering the string \( v^\min = v^\min_1 \cdots v^\min_n \), where each \( r^\min_i = \text{base}(r_i) \text{low}(r_i) \) and its counterpart \( w^\min = w^\min_1 \cdots w^\min_n \), where each \( w^\min_i = \text{base}(s_i) \text{low}(s_i) \), it is immediate that the sequence of non-zero lower bounds is the same in \( r' \) and \( s' \).

For the sake of a contradiction, let us now assume that there exists an \( i_{\min} \in \{1, \ldots, n\} \) with \( \text{low}(r_{i_{\min}}) < \text{low}(s_{i_{\min}}) \) and \( i_{\min} \) is minimal with this property. We consider two cases:

1) If \( \text{low}(r_{i_{\min}}) > 0 \), we define the string \( v' \) by replacing \( v^\max_{i_{\min}} \) in \( v^\max \) by \( \text{base}(r_{i_{\min}}) \text{low}(r_{i_{\min}}) \). As \( \text{low}(s_{i_{\min}}) > \text{low}(r_{i_{\min}}) \), this string cannot be matched by \( s' \). Indeed, as \( v' \) has \( n \) blocks, the only possible match for \( s' \) would match the \( i_{\min} \)-th block \( \text{base}(r_{i_{\min}}) \text{low}(r_{i_{\min}}) \) onto \( s_{i_{\min}} \). As this would mean that \( r \) is not equivalent to \( s \), this gives the desired contradiction.

2) In the second case, assume that \( 0 = \text{low}(r_{i_{\min}}) < \text{low}(s_{i_{\min}}) \). If \( \text{low}(s_{i_{\min}}) \geq 2 \), then the string resulting from \( w^\max \) by replacing \( w^\max_{i_{\min}} \) by the single symbol \( \text{base}(r_{i_{\min}}) \text{low}(r_{i_{\min}}) \) matches \( r' \) but does not match \( s' \), which again contradicts that \( r \) is equivalent to \( s \).

The only remaining case is that \( 0 = \text{low}(r_{i_{\min}}) < \text{low}(s_{i_{\min}}) = 1 \).

- If \( i_{\min} = 1 \), then the string \( v^\max_2 \cdots v^\max_n \) does not match \( s' \), as the string starts with the wrong symbol. However, the string matches \( r' \), which is a contradiction.

- If \( i_{\min} = n \), then the string \( v^\max_1 \cdots v^\max_{n-1} \) does not match \( s' \), as the string ends with the wrong symbol. However, the string matches \( r' \), which is a contradiction.

Hence, we know that \( 1 < i_{\min} < n \).

Let \( x \) be the string \( v^\max_1 \cdots v^\max_{i_{\min}-1} v^\max_{i_{\min}+1} \cdots v^\max_n \) which matches \( r' \) and therefore also matches \( s' \). If \( \text{base}(r_{i_{\min}-1}) \neq \text{base}(r_{i_{\min}+1}) \), then \( x \) has \( n - 1 \) blocks. Recall that for every \( j = 1, \ldots, n \), \( \text{base}(r_j) = \text{base}(s_j) \). As \( \text{base}(s_{i_{\min}}) \neq \text{base}(s_{i_{\min}+1}) \) and \( \text{base}(s_{i_{\min}}) \neq \text{base}(s_{i_{\min}-1}) \), \( s_{i_{\min}} \) can only match \( v^\max_j \) for some \( j > i_{\min} + 1 \) or \( j < i_{\min} - 1 \). But then all the blocks before \( v^\max_j \) or after \( v^\max_j \) (which are at least \( i_{\min} \) or \( n - i_{\min} + 1 \) blocks, respectively) must match \( s_1 \cdots s_{i_{\min}-1} \) or \( s_{i_{\min}+1} \cdots s_n \), respectively, which is impossible.

We are left with the case where \( \text{base}(r_{i_{\min}-1}) = \text{base}(r_{i_{\min}+1}) \).

(a) If \( r \) and \( s \) are from \( \text{RE}(a, a?) \) then neither \( s_{i_{\min}-1} \) nor \( s_{i_{\min}+1} \) matches \( v^\max_{i_{\min}-1} \cdots v^\max_{i_{\min}+1} \) as this string has length \( \text{upp}(r_{i_{\min}-1}) + \text{upp}(r_{i_{\min}+1}) \) = \( \text{upp}(s_{i_{\min}-1}) + \text{upp}(s_{i_{\min}+1}) \), which is more than \( \text{max} \text{max}(\text{upp}(s_{i_{\min}-1}), \text{upp}(s_{i_{\min}+1})) \).

As this would mean again that \( s_{i_{\min}} \) can only match \( v^\max_j \), for some
Figure 4: Graphical representation of several indices defined in the proof of Theorem 5.1

\[ j > i_{\min} + 1 \text{ or } j < i_{\min} - 1, \text{ we again have that } s \text{ can not match } x, \text{ a contradiction.} \]

(b) Now let \( r \) and \( s \) be from \( \text{RE}(a, a^*) \). Let \( j_{\min} > i_{\min} \) be minimal such that \( \text{low}(r_{j_{\min}}) \neq 0 \). As we already obtained that the sequence of non-zero lower bounds is the same in \( r' \) and \( s' \), such a \( j_{\min} \) must exist and \( \text{base}(r_{j_{\min}}) = \text{base}(r_{i_{\min}}) = \text{base}(s_{i_{\min}}) \). Let \( b \) be the symbol \( \text{base}(r_{j_{\min}}) = \text{base}(r_{i_{\min}}) \) and let \( a \) be the symbol \( \text{base}(r_{i_{\min} - 1}) = \text{base}(r_{i_{\min} + 1}) \). Notice that \( a \neq b \).

Our goal is to get a contradiction by showing that \( r' \) is not the strong sequence normal form of \( r \). On our way we have to deal with a couple of other possible cases.

Assume that there is some \( \ell, i_{\min} < \ell < j_{\min} \), with \( a \neq \text{base}(r_{\ell}) \neq b \). Let \( \ell' < i_{\min} \) be maximal such that \( a \neq \text{base}(r_{\ell'}) \neq b \). If there is no such \( \ell' \), we set \( \ell' = 0 \). We present the ordering of the defined indices \( i_{\min}, j_{\min}, \ell', \) and \( \ell \) in Figure 4.

Let, for each \( d = 1, \ldots, n \), \( x'_d \) be defined by

\[
x'_d = \begin{cases} 
\text{base}(r_d)^1 & \text{if low}(r_d) = 0, \\
\text{base}(r_d)^{\text{low}(r_d)} & \text{otherwise}.
\end{cases}
\]

Let \( x' = x'_1 \cdots x'_{\ell'} \cdot v_{\ell' + 1}^{i_{\min}} \cdots v_{\ell - 1}^{i_{\min}} x'_{\ell} \cdots x'_n \). Notice that the string \( v_{\ell' + 1}^{i_{\min}} \cdots v_{\ell - 1}^{i_{\min}} \) is non-empty. As \( x' \) matches \( r \), \( x' \) also matches \( s \). Note that, in this match, \( x'_{\ell'} \) has to be matched by \( s_{\ell'} \), because the string \( x'_1 \cdots x'_{\ell'} \) contains \( \ell' \) blocks. Analogously, \( x'_{\ell'} \) has to be matched by \( s_{\ell} \) because \( x'_{\ell} \cdots x'_n \) contains \( n - \ell + 1 \) blocks.

As \( i_{\min} \) is minimal such that \( \text{low}(r_{i_{\min}}) < \text{low}(s_{i_{\min}}) \) and as there are no factors \( f \) in \( r_{i_{\min}} \cdots r_{\ell} \) with \( \text{low}(f) > 0 \), we know that the number of factors \( f \) in \( s_{\ell' + 1} \cdots s_{\ell - 1} \) with \( \text{base}(f) = b \) and \( \text{low}(f) > 0 \) is larger than the number of such factors in \( r_{\ell' + 1} \cdots r_{\ell - 1} \). Hence, the number of \( b \)'s
in $x'$ between $x'_{\ell'}$ and $x'_\ell$ is smaller than the sum of the numbers low($f$) over the factors $f$ in $s_{\ell'+1} \cdots s_{\ell-1}$ with base($f$) = $b$. This contradicts the fact that $x'$ matches $s$.

We can conclude that there is no such $\ell$, that is, all factors between position $i_{\min}$ and $j_{\min}$ have symbol $a$ or $b$.

Let us consider next the possibility that the symbol base($r_{j_{\min}+1}$) exists and is different from $a$ and $b$, say base($r_{j_{\min}+1}$) = $c$. If low($s_{j_{\min}}$) = 0 then the string $x'_1 \cdots x'_{j_{\min}-1}x'_{j_{\min}+1} \cdots x'_n$ (consisting of $n - 1$ blocks, as $c$ is different from $a$ or $b$) matches $s$ but not $r$. This contradicts that $r$ and $s$ are equivalent. Let $\ell'$ be defined as before. If low($s_{j_{\min}}$) = 1 then the string $x'_1 \cdots x'_{\ell'}v_{\ell'+1} \cdots x'_{j_{\min}-1}x'_{j_{\min}+1} \cdots x'_n$ matches $r$ but not $s$ (as it has too few $b$s between $\ell'$ and $j_{\min}$). An analogous reasoning also works if $j_{\min} = n$.

We still need to deal with the case where base($r_{j_{\min}+1}$) = $a$. Hence, between position $i_{\min} - 1$ and $j_{\min} + 1$, $r'$ consists of a sequence of factors $f$ which alternate between base($f$) = $a$ and base($f$) = $b$, and ends with the factor $r_{j_{\min}+1}$ for which base($r_{j_{\min}+1}$) = $a$. Furthermore, all these factors, besides $r_{j_{\min}}$ and $r_{j_{\min}+1}$ have low($f$) = 0, and low($r_{j_{\min}}$) = 1. It follows immediately that $r'$ is not the sequence normal form of $r$ as it contains a subsequence of the form $a[i, *]b[0, *]a[0, *]b[1, *]a[l, *]$.

\[\square\]

6 Intersection

For arbitrary regular expressions, INTERSECTION is PSPACE-complete. We show that the problem is NP-hard for the same seemingly innocent fragments $RE(a, a^*)$, $RE(a, a?)$, $RE(a, (+a^*))$ and $RE(a^+, (+a))$ already studied in Section 4. By $RE(S - \{(+w)^*, (+w)^+\})$, we denote the fragment of $RE(S)$ where no factor can be of the form $(w_1 + \cdots + w_n)^*$ or $(w_1 + \cdots + w_n)^+$ with $n \geq 2$ and the length of at least one $w_i$ larger than two; so, factors of the form $(a_1 + \cdots + a_n)^*$ and $(a_1 + \cdots + a_n)^+$ are allowed. For the latter fragment, we obtain a matching NP-upper bound. INTERSECTION is already PSPACE-hard for $RE(a, (+w)^*)$ and $RE(a, (+w)^+)$ expressions. This follows from a proof of Bala, who showed that it is PSPACE-hard to decide whether the intersection of an arbitrary number of $RE((+w)^*)$ expressions contains a non-empty string [Bal02]. The precise complexity of $RE(a, w^+)$ remains open. These results are summarized in the following theorem:

**Theorem 6.1.** (a) INTERSECTION is NP-hard for

1. $RE(a, a^*)$;
2. $RE(a, a?)$;
3. $RE(a, (+a^*))$;

36
As in Section 4, we split the proof of Theorem 6.1 into a series of lemmas to improve readability.

**Proof of Theorem 6.1(a)** The proof is along the same lines as the proof of Lemma a. In all five cases, it is a logspace reduction from satisfiability of propositional 3CNF formulas. The satisfiability problem asks, given a propositional formula \( \Phi \) in 3CNF with variables \( \{x_1, \ldots, x_n\} \), whether there exists a truth assignment of \( \{x_1, \ldots, x_n\} \) for which \( \Phi \) is true. The satisfiability problem for 3CNF formulas is known to be \( \text{NP} \)–complete [GJ79]. We note that, for the cases (1)–(3), intersection is already \( \text{NP} \)-hard when the expressions use a fixed-size alphabet.

Let \( \Phi = C_1 \land \cdots \land C_k \) be a 3CNF formula using variables \( \{x_1, \ldots, x_n\} \). Let, for each \( i \), \( C_i = L_{i,1} \lor L_{i,2} \lor L_{i,3} \) be the \( i \)th clause with three literals. Our goal is to construct regular expressions \( R_1, \ldots, R_k \) and \( S_1, S_2 \) such that

\[
L(R_1) \cap \cdots \cap L(R_k) \cap L(S_1) \cap L(S_2) \neq \emptyset \quad \text{if and only if} \quad \Phi \text{ is satisfiable.}
\]

Analogously as in the proof of Lemma a, we encode truth assignments for \( \Phi \) by strings. We construct \( S_1 \) and \( S_2 \) such that \( L(S_1 \cap S_2) \) contains all such string representations \( w \) of truth assignments and a string \( w \) matches \( R_i \) if and only if \( w \) represents an assignment which makes \( C_i \) true.

We discuss the building blocks of these regular expressions. We make use of a regular expression \( U \) describing exactly one string \( w \): \( U \) will either be of the form \( a^n \) or of the form \( \#a^i\$ \cdots \$a^i\# \) (\( n \) occurrences of \( a^i \) separated by \( \$ \)) for some integer \( i \). We define expressions \( W_1 \) and \( W_2 \) such that each string \( w \) in \( L(W_1) \cap L(W_2) \) can be interpreted as a truth assignment. More precisely, for each truth assignment \( A \) there is a string \( w_A \in L(W_1) \cap L(W_2) \) and for each string \( w \in L(W_1) \cap L(W_2) \) there is a corresponding truth assignment \( A_w \).

We set

\[
S_1 = U^3W_1U^3, \\
S_2 = U^3W_2U^3, \quad \text{and} \\
R_i = NF_{i,1}F_{i,2}F_{i,3}N,
\]

for \( i = 1, \ldots, k \), where \( N \) and \( F_{i,1}, \ldots, F_{i,3} \) are regular expressions for which the following properties hold:
(i’) $u, u^2$, and $u^3 \in L(N)$.

(ii’) If $A$ makes $L_{i,j}$ true then $w_A \in L(F_{i,j}) \cap L(W_1) \cap L(W_2)$. If $w \in L(F_{i,j}) \cap L(W_1) \cap L(W_2)$ then $A_w$ makes $L_{i,j}$ true.

(iii’) $u \in L(F_{i,j})$ for every $i = 1, \ldots, k$ and $j = 1, 2, 3$.

(iv’) If $w^3w^3u^3 \in L(S_1) \cap L(S_2) \cap L(R_1) \cap \cdots \cap L(R_k)$, then $w$ matches some $F_{i,j}$, in every $R_i$.

We claim the following:

**Claim 6.2.** If there are expressions $U, W_1, W_2, N, F_{1,1}, F_{1,2}, F_{1,3}, F_{2,1}, \ldots, F_{k,1}, F_{k,2}, F_{k,3}$ satisfying (i’)-(iv’) above, then

$$L(S_1) \cap L(S_2) \cap L(R_1) \cap \cdots \cap L(R_k) \neq \emptyset$$

if and only if $\Phi$ is satisfiable.

**Proof.** Suppose that are expressions $U, W_1, W_2, N, F_{1,1}, F_{1,2}, F_{1,3}, F_{2,1}, \ldots, F_{k-1,1}, F_{k,2}, F_{k,3}$ satisfying (i’)-(iv’). We prove that $L(S_1) \cap L(S_2) \cap L(R_1) \cap \cdots \cap L(R_k) \neq \emptyset$ if and only if $\Phi$ is satisfiable.

$(\Rightarrow)$ Assume that $S_1 \cap S_2 \cap \bigcap_{i=1}^k R_i \neq \emptyset$. Hence, there exists a string $v = u^3w^3$ in $S_1 \cap S_2 \cap \bigcap_{i=1}^k R_i$, where $u$ is the unique string in $L(U)$. By (iv’), $w$ matches some $F_{i,j}$, in every $R_i$. By (ii’), $A_w$ makes $L_{i,j}$ true for every $i = 1, \ldots, k$. Hence, $\Phi$ is true under truth assignment $A_w$, so $\Phi$ is satisfiable.

$(\Leftarrow)$ Suppose now that $\Phi$ is true under some truth assignment $A$. Hence, for every $i$, some $L_{i,j}$ becomes true under $A$ and therefore $w_A \in L(F_{i,j})$ by (ii’). As $u, u^2,$ and $u^3$ are in $L(N)$ by (i’), and as $u$ is in each $L(F_{i,j})$ by (iii’), we get that the string $w^3w_Au^3$ is in $L(S_1) \cap L(S_2) \cap L(R_1) \cap \cdots \cap L(R_k)$. This completes the proof of the claim.

It remains to construct the regular expressions with the required properties. As in the proof of Lemma 5, we construct these expressions starting from five basic regular expressions $r_{\text{true}}, r_{\text{false}}, r_{\text{true,false}}, r_{\text{all}}$, and $\alpha$, which must adhere to the inclusion structure as shown in Figure 5 and formally defined by the properties (INT1)-(INT4) below. The two dots in Figure 5 denote the strings $z_{\text{true}}$ and $z_{\text{false}}$, which represent the truth values true and false, respectively. The expressions $r_{\text{true}}$ and $r_{\text{false}}$ are used to match $z_{\text{true}}$ and $z_{\text{false}}$, respectively, in the expressions $R_1, \ldots, R_k$. The expression $r_{\text{true,false}}$, is used to generate the truth values true and false. It will be defined as the intersection of two expressions in $S_1$ and $S_2$ and will be used to generate all truth assignments which must then be matched in $R_1, \ldots, R_k$. Finally, $\alpha$ and $r_{\text{all}}$ are used to ensure that condition (iv’) above holds. That is, they ensure that when $u^3w^3u^3 \in L(S_1) \cap L(S_2) \cap L(R_1) \cap \cdots \cap L(R_k)$, $w$ matches some $F_{i,j}$ in every $R_i$. 


Figure 5: Inclusion structure of regular expressions used in NP-hardness of intersection.

In all cases, these expressions have the properties INT1–INT4:

\[
\begin{align*}
\alpha & \in L(r^{\text{false}}) \cap L(r^{\text{true}}) \cap L(r^{\text{all}}) \quad \text{(INT1)} \\
z^{\text{false}} & \in L(r^{\text{false}}) \cap L(r^{\text{true},\text{false}}) \cap L(r^{\text{all}}) \quad \text{(INT2)} \\
z^{\text{true}} & \in L(r^{\text{true}}) \cap L(r^{\text{true},\text{false}}) \cap L(r^{\text{all}}) \quad \text{(INT3)} \\
L(r^{\text{false}}) \cap L(r^{\text{true}}) \cap L(r^{\text{true},\text{false}}) & = \emptyset \quad \text{(INT4)}
\end{align*}
\]

Note that \( z^{\text{false}} \not\in L(r^{\text{true}}) \) and \( z^{\text{true}} \not\in L(r^{\text{false}}) \) by (INT4). For the first three fragments, we now define the needed expressions. We deal with the other two fragments later. Note that the alphabet size is fixed (at most five) in the reduction for these fragments.

(1) For RE\((a, a^*)\):

- \( \alpha = a; \)
- \( r^{\text{true}} = aa^*b^*; \)
- \( r^{\text{false}} = b^*aa^*; \)
- \( r^{\text{true},\text{false}} = aa^*bb^* + bb^*aa^* \), which is constructed from the intersection of \( s^{\text{true},\text{false}} = b^*aa^*b^* \) and \( (s')^{\text{true},\text{false}} = a^*bb^*a^* \);  
- \( r^{\text{all}} = a^*b^*a^*; \)
- \( z^{\text{true}} = ab; \)
- \( z^{\text{false}} = ba; \)
- \( U = \#a\$ \cdots \#a\# \) (n occurrences of \( \alpha \));
- \( W_1 = \#s^{\text{true},\text{false}} \# \cdots \#s^{\text{true},\text{false}} \#; \)
- \( W_2 = \#(s')^{\text{true},\text{false}} \# \cdots \#(s')^{\text{true},\text{false}} \#; \)
- \( N = (\#^*a^*\$^*a^*\$^* \cdots \$^*a^*\$^*)^3 \) (n occurrences of \( a^* \) in each of the three copies).

(2) For RE\((a, a?)\):

39
• \( \alpha = a; \)
• \( r_{\text{true}} = ab; \)
• \( r_{\text{false}} = b?a; \)
• \( r_{\text{true,false}} = ab + ab, \) which is constructed from the intersection of
  \( s_{\text{true,false}} = b?ab? \) and \( (s')_{\text{true,false}} = a?ba?; \)
• \( r_{\text{all}} = a?b?a?; \)
• \( z_{\text{true}} = ab; \)
• \( z_{\text{false}} = ba; \)
• \( U = \#\alpha\# \cdot \cdots \cdot \#\alpha\# (n \text{ occurrences of } \alpha) \)
• \( W_1 = \#s_{\text{true,false}}\# \cdot \cdots \cdot \#s_{\text{true,false}}\#; \)
• \( W_2 = \#(s')_{\text{true,false}}\# \cdot \cdots \cdot \#(s')_{\text{true,false}}\#; \)
• \( N = (\#a?\#a?\# \cdot \cdots \cdot \#a?\#?)^3 (n \text{ occurrences of } a? \text{ in each of the}
  \text{ three copies}). \)

(3) For \( \text{RE}(a, (+a^+)); \)

• \( \alpha = a; \)
• \( r_{\text{true}} = (a + b)^+; \)
• \( r_{\text{false}} = (a + c)^+; \)
• \( r_{\text{true,false}} = (b + c)^+; \)
• \( r_{\text{all}} = (a + b + c)^+; \)
• \( z_{\text{true}} = b; \)
• \( z_{\text{false}} = c; \)
• \( U = \#\alpha\# \cdot \cdots \cdot \#\alpha\# (n \text{ occurrences of } \alpha); \)
• \( W_1 = W_2 = \#r_{\text{true,false}}\# \cdot \cdots \cdot \#r_{\text{true,false}}\#; \) and
• \( N = (\# + \# + a)^+ \)

It is easy to check that the conditions (INT1)–(INT4) are fulfilled for all of the
fragments.

With \( w = \#w_1\# \cdot \cdots \cdot \#w_n\# \in L(W_1) \cap L(W_2) \) we associate a truth assignment
\( A_w \) as follows

\[
A_w(x_j) := \begin{cases} 
  \text{true}, & \text{if } w_j \in L(r_{\text{true}}), \\
  \text{false}, & \text{otherwise}.
\end{cases}
\]

Let \( z_{\text{false}} \in L(r_{\text{false}}) \cap L(r_{\text{all}}) \cap L(r_{\text{true,false}}) \) and \( z_{\text{true}} \in L(r_{\text{true}}) \cap L(r_{\text{all}}) \cap L(r_{\text{true,false}}). \) By (INT2) and (INT3) we know that these strings exist. Notice
that \( z_{\text{false}} \not\in L(r_{\text{true}}) \) and \( z_{\text{true}} \not\in L(r_{\text{false}}) \) by (INT4). For a truth assignment
\( A, \) let

\[
w_A = \#w_1\# \cdot \cdots \cdot \#w_n\#, \]

where \( w_j = z_{\text{true}} \) if \( A(x_j) = \text{true} \) and \( w_j = z_{\text{false}}, \) otherwise.
For each \( i, j \), we set
\[
F_{i,j} = \#e_1\$ \cdots \#e_n\#,
\]
where for each \( \ell = 1, \ldots, n \),
\[
e_\ell := \begin{cases} r^{\text{false}}, & \text{if } L_{i,j} = \neg x_\ell, \\ r^{\text{true}}, & \text{if } L_{i,j} = x_\ell, \\ r^{\text{all}}, & \text{otherwise}. \end{cases}
\]
Notice that only one \( e_\ell \) among \( \{e_1, \ldots, e_n\} \) is different from \( r^{\text{all}} \).

It remains to show that, for each of the fragments, conditions (i')–(iv') hold.

(i') Trivial.

(ii') Let \( A \) be a truth assignment such that \( L_{i,j} \) is true under \( A \). We need to show that \( w_A \in L(F_{i,j}) \cap L(W_1) \cap L(W_2) \). Suppose that \( L_{i,j} = x_\ell \), so, the variable \( x_\ell \) occurs positively in clause \( C_\ell \) and \( A(x_\ell) = \text{true} \). Let \( F_{i,j} = \#e_1\$ \cdots \#e_n\# \) be as defined above. By definition of \( F_{i,j} \), we have that \( e_\ell = r^{\text{true}} \) and for all \( \ell' \neq \ell \) we have that \( e_{\ell'} = r^{\text{all}} \). Let \( w_A = \#w_1\$ \cdots \#w_n\# \) be as defined above, so \( w_\ell = z_{\ell}^{\text{true}} \) by definition of \( w_A \). Notice that, for every \( \ell' \neq \ell \), \( w_{\ell'} \) is either \( z_{\ell'}^{\text{false}} \) or \( z_{\ell'}^{\text{true}} \). Since \( z_{\ell}^{\text{true}} \in L(r_{\ell}^{\text{true}}) \cap L(r_{\ell}^{\text{all}}) \) and \( \{z_{\ell}^{\text{true}}, z_{\ell}^{\text{false}}\} \subseteq L(r_{\ell}^{\text{all}}) \), we have that \( w_A \in L(F_{i,j}) \). Moreover, as (INT2) and (INT3) state that \( \{z_{\ell}^{\text{true}}, z_{\ell}^{\text{false}}\} \subseteq L(r_{\ell}^{\text{true,false}}) \), we also have that \( w_A \in L(W_1) \cap L(W_2) \). The dual statement also holds if \( L_{i,j} = \neg x_\ell \).

For the second claim, let \( w = \#w_1\$ \cdots \#w_n\# \) be a string in \( L(F_{i,j}) \cap L(W_1) \cap L(W_2) \), where \( F_{i,j} = \#e_1\$ \cdots \#e_n\# \). We now need to show that \( L_{i,j} \) is true under truth assignment \( A_w \). If \( L_{i,j} = x_\ell \), then we have that \( e_\ell = r_{\ell}^{\text{true}} \) by definition of \( F_{i,j} \). As \( w \) matches \( F_{i,j} \), we have that \( w_\ell \in L(r_{\ell}^{\text{true}}) \). Hence, by definition of \( A_w \), \( A_w(x_\ell) = \text{true} \), so \( L_{i,j} \) is true under \( A_w \). If \( L_{i,j} = \neg x_\ell \), then we have that \( e_\ell = r_{\ell}^{\text{false}} \). This means that \( w_\ell \in r_{\ell}^{\text{true,false}} \cap r_{\ell}^{\text{false}} \), as \( w \) is also in \( L(W_1) \cap L(W_2) \). Consequently, \( w_\ell \not\in r_{\ell}^{\text{true}} \) by (INT4). By definition of \( A_w \), we now have that \( A_w(x_\ell) = \text{false} \), and \( A_w \) again makes \( L_{i,j} \) true.

(iii') This follows immediately from condition (INT1).

(iv') Suppose that \( w^3 w u^3 \in L(S_1) \cap L(S_2) \cap L(R_1) \cap \cdots \cap L(R_k) \). We need to show that \( w \) matches some \( F_{i,j} \), in every \( R_i \). Observe that the string \( w^3 w u^3 \) is of the form \( \#y_1\# \#y_2\# \#y_3\# \#y_4\# \#y_5\# \#y_6\# \#y_7\# \), where \( y_1, \ldots, y_7 \) are non-empty strings over the alphabet \( \{a, b, c, \$\} \). Moreover, every string in every \( L(F_{i,j}) \) is of the form \( \#y\# \) where \( y \) is a non-empty string over the alphabet \( \{a, b, c, \$\} \). Hence, as for every \( i = 1, \ldots, k \), \( w^3 w u^3 \in L(R_i) \), and as none of the strings \( y, y_1, \ldots, y_7 \) contain the symbol “\#”, \( w \) either matches some \( F_{i,j} \) or \( w \) matches a sub-expression of \( N \).
Fix an $i = 1, \ldots , k$. Let $m$ be a match between $w^3 w w^3$ and $R_i$. Let $\ell$ be the number so that $m$ matches $\# y_1 \# \cdots \# y_\ell \#$ onto the left occurrence of $N$ in $R_i$. We now distinguish between fragments (1–2) and fragment (3).

- In fragments (1–2) we have that $\ell \leq 3$. Towards a contradiction, assume that $m$ matches $w$ onto the left occurrence of $N$ in $R_i$. But this would mean that $m$ matches a superstring of $\# y_1 \# \cdots \# y_4 \#$ onto this leftmost occurrence of $N$, which contradicts that $\ell \leq 3$. Analogously, $m$ cannot match $w$ onto the right occurrence of $N$ in $R_i$. So, $w$ must match by some $F_{i,j}$ for $j = 1, 2, 3$.

- In fragment (3), we have that $\ell \geq 1$. Again, towards a contradiction, assume that $m$ matches $w$ onto the right occurrence of $N$ in $R_i$. Observe that any string that matches $N F_{i,1} F_{i,2} F_{i,3}$ is of the form $\# y_1 \# \# y_2 \# \cdots \# y_\ell \#$, where $\ell' > 3$. As $u^3$ is not of this form, $m$ cannot match $u^3$ onto $N F_{i,1} F_{i,2} F_{i,3}$, which is a contradiction. Analogously, $m$ cannot match $w$ against the left occurrence of $N$ in $R_i$. So, $m$ must match $w$ onto some $F_{i,j}$ for $j = 1, 2, 3$.

This completes the proof of Lemma a for the fragments (1)–(3).

We still need to deal with the fragments (4) and (5). As in the proof of Lemma a, we will no longer use an alphabet with fixed size. Instead, we use the symbols $b_j$ and $c_j$ for $j = 1, \ldots , n$. Instead of the basic regular expressions $r_{\text{true}}$, $r_{\text{false}}$, $r_{\text{true,false}}$, and $r_{\text{all}}$, we will now have expressions $r_{\text{true}}^j$, $r_{\text{false}}^j$, $r_{\text{true,false}}^j$, and $r_{\text{all}}^j$ for every $j = 1, \ldots , n$. We will require that these expressions have the same properties (INT1)–(INT4), but only between the expressions $r_{\text{true}}^j$, $r_{\text{false}}^j$, $r_{\text{true,false}}^j$, and $r_{\text{all}}^j$ with the same index $j$, and $\alpha$.

The needed regular expressions are then defined as follows:

(4) For RE($a, (+a)^+$):

- $\alpha = a$;
- $r_{\text{true}}^j = (a^+ + b_j^+)$;
- $r_{\text{false}}^j = (a^+ + c_j^+)$;
- $r_{\text{true,false}}^j = (b_j^+ + c_j^+)$;
- $r_{\text{all}}^j = (a^+ + b_j^+ + c_j^+)$;
- $z_{\text{true}} = b_j$;
- $z_{\text{false}} = c_j$;
- $U = \alpha^n$;
- $W_1 = W_2 = r_{\text{true,false}}^1 \cdots r_{\text{true,false}}^n$; and
- $N = a^+$

(5) For RE($a^+, (+a)$):
\(\alpha = a\);
\(r^\text{true}_j = (a + b_j)\);
\(r^\text{false}_j = (a + c_j)\);
\(r^\text{true, false}_j = (b_j + c_j)\);
\(r^\text{all}_j = (a + b_j + c_j)\);
\(z^\text{true} = b_j\);
\(z^\text{false} = c_j\);
\(U = \alpha^n\);
\(W_1 = W_2 = r^\text{true, false}_1 \cdots r^\text{true, false}_n\); and
\(N = a^+\)

With \(w = \# w_1 \# \cdots \# w_n \# \in L(W_1) \cap L(W_2)\) we associate a truth assignment \(A_w\) as follows
\[
A_w(x_j) := \begin{cases} 
\text{true,} & \text{if } w_j \in L(r^\text{true}_j), \\
\text{false,} & \text{otherwise.}
\end{cases}
\]

Let \(z^\text{false}_j \in L(r^\text{false}_j) \cap L(r^\text{all}_j) \cap L(r^\text{true, false}_j)\) and \(z^\text{true}_j \in L(r^\text{true}_j) \cap L(r^\text{all}_j) \cap L(r^\text{true, false}_j)\). By (INT2) and (INT3) we know that these strings exist. Notice that \(z^\text{false}_j \not\in L(r^\text{true}_j)\) and \(z^\text{true}_j \not\in L(r^\text{false}_j)\) by (INT4). For a truth assignment \(A\), let
\[
w_A = \# w_1 \# \cdots \# w_n \#,
\]
where \(w_j = z^\text{true}_j\) if \(A(x_j) = \text{true}\) and \(w_j = z^\text{false}_j\), otherwise.

For each \(i, j\), we set
\[
F_{i,j} = \# e_1 \# \cdots \# e_n \#,
\]
where for each \(\ell = 1, \ldots, n\),
\[
e_\ell := \begin{cases} 
\text{false,} & \text{if } L_{i,j} = \neg x_\ell, \\
\text{true,} & \text{if } L_{i,j} = x_\ell, \text{ and} \\
\text{all,} & \text{otherwise.}
\end{cases}
\]

It remains to show that, for each of the fragments, conditions (i')–(iv') hold.

(i') Trivial.

(ii') This can be shown analogously as for the fragments (1)–(3).

(iii') This follows immediately from condition (INT1).

(iv') Suppose that \(w^3 w u^3 \in L(S_1) \cap L(S_2) \cap L(R_1) \cap \cdots \cap L(R_k)\). We need to show that \(w\) matches some \(F_{i,j}\) in every \(R_i\). To this end, let \(m_i\) be a match between \(w^3 w u^3\) and \(R_i\), for every \(i = 1, \ldots, k\). For every \(j = 1, \ldots, n\), let \(\Sigma_j\) denote the set \(\{b_j, c_j\}\). Observe that the string \(w\) is of the form \(y_1 \cdots y_n\), where, for every \(j = 1, \ldots, n\), \(y_j\) is a string in \(\Sigma_j^+\). Moreover,
no strings in $L(N)$ contain symbols from $\Sigma_j$ for any $j = 1, \ldots, n$. Hence, $m_i$ cannot match any symbol of the string $w$ onto $N$. Consequently, $m_i$ matches the entire string $w$ onto a subexpression of $F_{i,1}F_{i,2}F_{i,3}$ in every $R_i$.

Further, observe that every string in every $F_{i,j}$, $j = 1, 2, 3$, is of the form $y_1' \cdots y_n'$, where each $y_j'$ is a string in $(\Sigma_j \cup \{a\})^+$. As $m_i$ can only match symbols in $\Sigma_j$ onto subexpressions with symbols in $\Sigma_j$, $m_i$ matches $w$ onto some $F_{i,j}$.

This completes the proof of Lemma a. □

The following Lemma makes the difference between INCLUSION and INTERSECTION apparent. Indeed, INCLUSION for $\text{RE}(S-\{(+w)^*, (+w)^{+}\})$ expressions is PSPACE-complete, while INTERSECTION for such expressions is NP-complete. The latter, however, does not imply that INCLUSION is always harder than INTERSECTION. Indeed, we obtained that for any fixed $k$, INCLUSION for $\text{RE}^{\leq k}$ is in PTIME. Later in this section, we will show that INTERSECTION is PSPACE-hard for $\text{RE}^{\leq k}$ expressions (Theorem 6.4).

**Proof of Theorem 6.1(b)** Let $r_1, \ldots, r_k$ be $\text{RE}(S-\{(+w)^*, (+w)^{+}\})$ expressions. We prove that if $\bigcap_{i=1}^{k} L(r_i) \neq \emptyset$, then the shortest string $w$ in $\bigcap_{i=1}^{k} L(r_i)$ has a representation as a compressed string of polynomial size. The idea is that the factors of the expressions $r_i$ induce a partition of $w$ into at most $kn$ substrings, where $n = \max\{|r_i| \mid 1 \leq i \leq k\}$. We show that each such substring is either short or it is matched by an expression of the form $w^*$ or $w^\dagger$. In the latter case, this substring can be written as $(x,1)(w,i)(y,1)$, for a suitable $i$, where $x$ and $y$ are a suffix and prefix of $w$, respectively. This immediately implies the statement of the theorem, as guessing $v$ and verifying that string($v$) is in each $L(r_i)$ is an NP algorithm by Lemma 2.3.

For simplicity, we assume that all $r_i$ have the same number of factors, say $n'$. Otherwise, some expressions can be padded by additional factors $\varepsilon$. Let, for each $i$, $r_i = e_{i,1} \cdots e_{i,n'}$, where every $e_{i,j}$ is a factor.

Let $u = a_1 \cdots a_{\text{min}}$ be a minimal string in $\bigcap_{i=1}^{k} r_i$. We will show that there is a polynomial size compressed string $v$ such that string($v$) = $u$. As the straightforward nondeterministic product automaton for $\bigcap_{i=1}^{k} r_i$ has at most $n^k$ states, $|u| \leq n^k$.

Let, for each $i = 1, \ldots, k$, $m_i$ be a match between $u$ and $r_i$. Notice that, for each $i = 1, \ldots, k$ and $j = 1, \ldots, n'$, there is exactly one pair $(\ell, \ell')$ such that $e_{i,j} \in m_i(\ell, \ell')$. We call an interval $(p, p')$ of positions homogeneous, if, for each $i$, there are $\ell, \ell'$ and $j$ such that $\ell \leq p$, $p' \leq \ell'$ and $e_{i,j} \in m_i(\ell, \ell')$. Intuitively, an interval is homogeneous if, for each $i$, all its symbols are subsumed by the same subexpression $e_{i,j}$. We call an interval $(p, p')$ maximally homogeneous, if $(p, p')$ is homogeneous, but $(p, p' + 1)$ and $(p - 1, p')$ are not homogeneous.

Let $\ell_0, \ldots, \ell_{\text{maxpos}}$ be a non-decreasing sequence of positions of $u$ such that $\ell_0 = 0$, $\ell_{\text{maxpos}} = \min$, and each pair $(\ell_p + 1, \ell_{p+1})$ is maximally homogeneous. Notice that $\ell_0, \ldots, \ell_{\text{maxpos}}$ maximally contains $kn'$ positions.
Let, for each $p = 1, \ldots, \min$, $u_p$ denote the substring of $u$ from position $\ell_{p-1}+1$ until position $\ell_p$. We consider each substring $u_p$ separately and distinguish the following cases:

- If $u_p$ is contained in an interval which at least one $m_i$ matches onto an expression of the form $e$ or $e?$ for a disjunction of base symbols $e$ which is (in abbreviated notation) of the form $a, w, (+a),$ or $(+w)$, then $|u_p| \leq |e| \leq n$. We set $v_p = u_p$.

- If $u_p$ is only contained in intervals which are mapped to expressions of the form $(+a)^*$ or $(+a)^+$ (in abbreviated notation) then $u_p$ has length zero or one. Otherwise, deleting a symbol of $u_p$ in $u$ would result in a shorter string $u'$ which is still in every $L(r_p)$, which contradicts that $u$ is a minimal string in $\bigcap_{i=1}^{k} L(r_i)$.

- The only remaining case is that $u_p$ is contained in some interval which at least one $m_i$ matches onto an expression of the form $a^*$, $w^*$, $a^+$, $w^+$, $(+a^*)$, $(+w^*)$, $(+a^+)$ or $(+w^+)$. Hence, $u_p$ can be written as $xw^iy$ for some $i$, a postfix $x$ of $w$ and a prefix $y$ of $w$. Notice that the length of $x$ and $y$ is at most $n$. We define $v_p = (x, 1)(w, i)(y, 1)$. Of course, $i \leq n^k$, hence $|v_p| = O(n)$.

Finally, let $v = v_1 \cdots v_{\min}$. Hence, $v$ is a compressed string with $\text{string}(v) = u$ and $|v| = O(kn' \cdot n)$, as required. □

Bala has shown that deciding whether the intersection of $\text{RE}(+(w)^*)$ expressions contains a non-empty string is PSPACE-complete [Bal02]. In general, the intersection problem for such expressions is trivial, as the empty string is always in the intersection. We next present a significantly simplified proof of the result of Bala (which is a direct reduction from acceptance by a Turing machine that uses polynomial space), adapted to show PSPACE-hardness of INTERSECTION for $\text{RE}(a, (+w)^*)$ and $\text{RE}(a, (+w)^{+})$ expressions. It should be noted that the crucial idea in this proof comes from the proof of Bala [Bal02].

**Proof of Theorem 6.1(c)** We first show that intersection is PSPACE-hard for $\text{RE}(a, (+w)^+)$ and we consider the case of $\text{RE}(a, (+w)^*)$ later. In both cases, we use a reduction from CORRIDOR TILING, which is PSPACE-complete [Chl86].

To this end, let $D = (T, H, V, \overline{b}, \overline{t}, n)$ be a tiling system. Without loss of generality, we assume that $n \geq 2$ is an even number. We construct $n+3$ regular expressions $\text{BT}$, $\text{Hor}_{\text{even}}$, $\text{Hor}_{\text{odd}}$, $\text{Ver}_1, \ldots, \text{Ver}_n$ such that

$$L(\text{BT}) \cap L(\text{Hor}_{\text{even}}) \cap L(\text{Hor}_{\text{odd}}) \cap \bigcap_{j=1}^{n} L(\text{Ver}_j) \neq \emptyset$$

if and only if there exists a correct corridor tiling for $D$.

Let $T = \{t_1, \ldots, t_k\}$ be the set of tiles of $D$. In our regular expressions, we will use a different alphabet for every column of the tiling. To this end, for every
For a set of $\Sigma$-symbols $S$, we denote by $S$ the disjunction of all the symbols of $S$, whenever this improves readability.

We represent possible tilings of $D$ as strings in the language defined by the regular expression

\[
\delta((\Delta^n \Sigma_1 \Delta^n \Sigma_2 \Delta^n \cdots \Delta^n \Sigma_n \Delta^n \#)^*)<, \quad (\star)
\]

where $\delta$, $<$, $\Delta$, and $\#$ are special symbols not occurring in $\Sigma$. Here, we use the symbols “$\delta$” and “$<$” as special markers to indicate the begin and the end of the tiling, respectively. Furthermore, the symbol “$\#$” is a separator between successive rows. The symbol “$\Delta$” is needed to enforce the vertical constraints on strings that represent tilings, which will become clear later in the proof. It is important to note that we do not use the regular expression $(\star)$ itself in the reduction, as it is not a RE($a, (+w)^+$) expression.

We are now ready for the reduction. We define the necessary regular expressions. Let $\overline{b} = (\text{bot}_1, \ldots, \text{bot}_n)$ and $\overline{t} = (\text{top}_1, \ldots, \text{top}_n)$.

- The following RE($a, (+w)^+$) expression ensures that the tiling begins and ends with the bottom and top row, respectively:

\[
\begin{align*}
\text{BT} := & \; \delta \Delta^n \text{bot}_{1,1} \Delta^n \cdots \Delta^n \text{bot}_{n,n} \Delta^n \\
& (\Sigma \cup \{\#, \Delta\})^+ \Delta^n \text{top}_{1,1} \Delta^n \cdots \Delta^n \text{top}_{n,n} \Delta^n \# <
\end{align*}
\]

- The following expression verifies the horizontal constraints between tiles in columns $\ell$ and 1, where $\ell$ is an even number between 1 and $n$. Together with Hor$_{\text{odd}}$, this expression also ensures that the strings in the intersection are correct encodings of tilings. (That is, the strings are in [Equation].)

\[
\text{Hor}_{\text{even}} := \left( \sum_{2 \leq \ell \leq n-2 \atop \ell \text{ even}} \sum_{(t_i, t_j) \in \mathcal{V}} \left( \Delta^n t_{i, \ell} \Delta^n t_{j, \ell+1} \right) \right) + \left( \delta \Delta^n \text{bot}_{1,1} \right) + \left( \delta \Delta^n \text{top}_{n,n} \Delta^n \# < \right) + \sum_{\substack{1 \leq i \leq k \atop 1 \leq j \leq k}} \left( \Delta^n t_{i,n} \Delta^n \# \Delta^n t_{j,1} \right).
\]

The last three disjuncts take care of the very first, very last and all start and end tiles of intermediate rows.

- The following expression verifies the horizontal constraints between tiles in columns $\ell$ and $\ell + 1$, where $\ell$ is an odd number between 1 and $n - 1$. Together with Hor$_{\text{even}}$, this expression also ensures that the strings in the intersection are correct encodings of tilings. (That is, the strings are in [Equation].)
the language defined by \((\ast)\).

\[
\text{Hor}_{\text{odd}} := (\Delta^n\text{bot}_{1,1}\Delta^n\text{bot}_{2,2}) + (\Delta^n\text{top}_{n-1,n-1}\Delta^n\text{top}_{n,n}\Delta^n\#<) + \sum_{1 \leq i \leq n-1 \atop \ell \text{ odd} \atop (t_i, t_j) \in V} (\Delta^n t_{1,\ell} \Delta^n t_{j,\ell+1})^+ + (\Delta^n\#)^+ .
\]

- Finally, for each \(\ell = 1, \ldots, n\), the following expressions verify the vertical constraints in column \(\ell\).

\[
\text{Ver}_\ell := \left( \sum_{(\text{bot}_\ell, t_i) \in V} (\Delta^n\text{bot}_{1,1}\Delta^n\cdots\Delta^n\text{bot}_{\ell,\ell}) \right) + \sum_{1 \leq i \leq n \atop m \neq \ell} (\Delta^i t_{j,m} \Delta^n - i) + \sum_{1 \leq i \leq n} (\Delta^i\#\Delta^n - i) + \sum_{1 \leq i \leq n} (\Delta^i\#<) + \sum_{1 \leq i \leq n \atop (t_i, t_j) \in V} (\Delta^i t_{i,\ell} \Delta^n - j)^+ .
\]

The crucial idea is that, when we read a tile in the \(\ell\)-th column, we use the string \(\Delta^n - i\) to encode that we want the tile in the \(\ell\)-th column of the next row to be \(t_i\). By using the disjunctions over the vertical constraints, we can express all the possibilities of the tiles that we allow in the next column.

We explain the purpose of the five disjunctions in the expression. We can assume here that the strings are already in the intersection of Hor_{odd} and Hor_{even}. This way, we know that \(i\) the strings encode tilings in which every row consists of \(n\) tiles, \(ii\) we use symbols from \(\Sigma_j\) to encode tiles from the \(j\)-th column, and that \(iii\) the string \(\Delta^n\) occurs between every two tiles. The first disjunction allows us to get started on the bottom row. It says that we want the \(\ell\)-th tile in the next row to be a \(t_i\) such that \((\text{bot}_\ell, t_i) \in V\). The second, and third disjunction ensure that this number \(i\) is “remembered” when reading \((a)\) tiles that are not on the \(\ell\)-th column or \((b)\) the special delimiter symbol “\#” which marks the beginning of the next row. The fourth disjunction can only be matched at the end of the tiling, as the symbol \(<\) only occurs once in each correctly encoded tiling. Finally, the fifth disjunction does the crucial step: it ensures that, when we matched the \(\ell\)-th tile \(t_k\) in the previous row with an expression ending with \(t_{k,\ell} \Delta^n - i\), and we remembered the number \(i\) when matching all the tiles up to the current tile, we are now obliged to use a disjunct of the form \(\Delta^i t_{i,\ell} \Delta^n - j\) to match the current tile. In particular, this means that the current tile must be \(t_i\), as we wanted. Further, the disjunction over \((t_i, t_j) \in V\) ensures again that the \(\ell\)-th tile in the next column will be one of the \(t_j\)'s.
It is easy to see that a string \( w \) is in the intersection of \( BT, \) Hor\(_{\text{even}}\), Hor\(_{\text{odd}}\), Ver\(_1\), \ldots, Ver\(_n\), if and only if \( w \) encodes a correct tiling for \( D \).

The \( \text{pspace} \)-hardness proof for \( \text{RE}(a, (+w)^*) \) is completely analogous, except that every “+” (which is not a disjunction) needs to be replaced by a “*”.

\[ \square \]

Theorem 6.1(d) holds as intersection for arbitrary regular expressions is in \( \text{pspace} \). This concludes the proof of Theorem 6.1.

We recall the notion of one-unambiguous regular expressions [BKW98]. A marking of a regular expression \( r \) is an assignment of different subscripts to every alphabet symbol occurring in \( r \). More formally, the marking of a regular expression \( r \), denoted by \( \text{mark}(r) \), is obtained from \( r \) by replacing the \( i \)-th alphabet symbol \( a \) in \( r \) by \( a_i \) (counting from left to right). For instance, the marking of \( (a+b)^*ab \) is \( (a_1+b_2)^*a_3b_4 \). For \( w \in L(\text{mark}(r)) \), we denote by \( w^\# \) the string obtained from \( w \) by dropping the subscripts. For instance, when \( w = a_1a_3b_4 \), then \( w^\# = aab \).

**Definition 6.3.** A regular expression \( r \) is one-unambiguous if for all strings \( u, v, w \) and symbols \( x, y \), the conditions \( uxv, uyw \in L(\text{mark}(r)) \) and \( x \neq y \) imply that \( x^\# \neq y^\# \).

For instance, \( r_1 = (a+b)(a+b)^* \) is a one-unambiguous regular expression, while \( r_2 = (a+b)^*(a+b) \) is not. Indeed, we have that \( \text{mark}(r_2) = (a_1+b_2)^*(a_3+b_4) \) and that \( a_1a_3 \in \text{mark}(r_2) \) and \( a_1a_1a_3 \in \text{mark}(r_2) \). If we take \( u = a_1, x = a_3, v = \varepsilon, y = a_1 \) and \( w = a_3 \), we see that \( x \neq y \) and \( x^\# = y^\# \), which contradicts the definition.

**Theorem 6.4.** Intersection is \( \text{pspace} \)-complete for

(a) one-unambiguous regular expressions; and for

(b) \( \text{RE} \leq^3 \).

**Proof.** The \( \text{pspace} \) upper bound is immediate as intersection is in \( \text{pspace} \) for regular expressions in general.

We proceed by showing the lower bound. It is relatively straightforward to reduce Corridor Tiling to these problems. The result for (a) is obtained by defining the regular expressions in such a way that they use separate alphabets for the last rows of the tiling and by using separate alphabets for odd and even rows. The result for (b) is obtained by carefully defining the reduction such that each symbol in the expressions is only used for positions with the same offset in a tiling row and the same parity of row number.

We reduce Corridor Tiling, which is \( \text{pspace} \)-complete [Chl86], to both intersection problems. We first show that intersection is \( \text{pspace} \)-hard for one-unambiguous regular expressions and then adapt these expressions so that they only contain up to three occurrences of the same alphabet symbol.

Let \( D = (T, H, V, 7, 7, n) \) be a tiling system. Without loss of generality, we assume that every correct tiling has an even number of rows and that the tiles
are partitioned into three disjoint sets $T_0 \cup T' \cup T''$. The idea is that symbols from $T''$, and $T'$ are only used on the uppermost and one but uppermost row, respectively. We denote symbols from $T'$ and $T''$ with one and two primes, respectively. The assumption for an even number of rows simplifies the expressions that check the vertical constraints.

From $D$ we construct regular expressions $BT$, $Hor_{t,i}$, $Ver$-odd$_{t,j}$, $Ver$-even$_{t,j}$, and Ver-last$_{t,j}$ for every $t \in T$, $i \in \{1, \ldots, n-1, n+1, \ldots, 2n-1\}$, and $j \in \{1, \ldots, n\}$. These expressions are constructed such that, for every string $w$, we have that

$$w \in L(BT) \cap \bigcap_{t,i} L(Hor_{t,i}) \cap \bigcap_{t,j} (L(Ver$-odd$_{t,j}) \cap L(Ver$-even$_{t,j}) \cap L(Ver$-last$_{t,j}))$$

if and only if $w$ encodes a correct tiling for $D$. ($\Diamond$)

For a set of tiles $S$, we sometimes denote by $S$ the disjunction of all symbols in $S$ whenever this improves readability. We also write $S^i$ for a sequence of $i$ times $S$. We define the following regular expressions:

- Let $\overline{b} = b_1 \cdots b_n$ and $\overline{t} = t'_1 \cdots t''_n$. Then the expression

$$BT := b_1 \cdots b_n \left( T_n^0 \left( T_n^0 + (T')^n \right) \right)^* t'_1 \cdots t''_n$$

expresses that (i) the tiling consists of an even number of rows, (ii) the first row is tiled by $\overline{b}$, and (iii) the last row is tiled by $\overline{t}$. It is easy to see that this expression is one-unambiguous.

- The following expressions verify the horizontal constraints. For each $t \in T$ and each $j \in \{1, \ldots, n-1, n+1, \ldots, 2n-1\}$, the expression $Hor_{t,j}$ ensures that the right neighbour of $t$ is correct, where $t$ is a tile

- in the $j$-th column when $j \leq n-1$, or
- in the $(j - n)$-th column when $j > n$.

Formally, we define for each $t \in T$ and each $j \in \{1, \ldots, n-1, n+1, \ldots, 2n-1\}$ the expression

$$Hor_{t,j} := \left( T_j^{-1} (t(s_1 + \cdots + s_t) + (\{T - \{t\}\}T)T^{2n-j-1}) \right)^* ,$$

where $s_1, \ldots, s_t$ are all tiles $s_i$ with $(t, s_i) \in H$. This expression is one-unambiguous.

- Correspondingly, we have expressions for the vertical constraints. For each $t$ and each $j \in \{1, \ldots, n\}$, there are three kinds of expressions checking the vertical constraints in the $j$-th column for the tile $t$: $Ver$-odd$_{t,j}$ checks the vertical constraints on all odd rows but the last, $Ver$-even$_{t,j}$ checks the vertical constraints on the even rows, and $Ver$-last$_{t,j}$ checks the vertical
constraints on the last two rows. We assume that \( \{s_1, \ldots, s_\ell\} \) is the set of tiles \( s_i \) with \( (t, s_i) \in V \).

We define the expressions Ver-odd\(_{t,j}\) formally as follows:

\[
\text{Ver-odd}_{t,j} := T_0^{-j-1} \\
\left[ (t T_0^n (s_1 + \cdots + s_\ell)) + ((T_0 - \{t\}) T_0^n) \right] T_0^{n-j} (T_0^{j-1} + T'^{j-1})^* T^{m-n-j+1} T'^{m-n}
\]

Intuitively, every occurrence of the tile \( t \) in the \( j \)-th column of any odd row (except the last odd row) of a tiling matches the leftmost \( t \) in Ver-odd\(_{t,j}\). The \( n \)-th tile starting from \( t \) then has to match the disjunction \( (s_1 + \cdots + s_\ell) \), which ensures that the vertical constraint is satisfied. If the tile in the \( j \)-th column is different from \( t \) (which is handled by the subexpression \( (T_0 - \{t\}) \)), the expression allows every tile in the \( j \)-th column in the next row. Further, it is easy to see that every string matching the subexpression in the Kleene star has length \( 2n \), and that the expression Ver-odd\(_{t,j}\) is one-unambiguous.

We define the expressions Ver-even\(_{t,j}\) formally as follows:

\[
\text{Ver-even}_{t,j} := T_0^n T_0^{j-1} \\
\left[ (t T_0^{-j-1} (s_1 + \cdots + s_\ell)) + ((T_0 - \{t\}) T_0^n) \right] (T_0^{n-j} (T_0^{j-1} + T'^{j-1})^* (T_0^{m-n-j} T'^{m-n-j+1} T'^{m})
\]

Here, every occurrence of the tile \( t \) in the \( j \)-th column of any even row (except the last row) of a tiling matches the leftmost \( t \) in Ver-odd\(_{t,j}\). Depending whether the \( n \)-th tile starting from \( t \) is on the row tiles with \( T' \) or not, the tile then either has to match the disjunction \( (s_1 + \cdots + s_\ell) \) or \( (s'_1 + \cdots + s'_\ell) \). This ensures that the vertical constraint is satisfied. If the tile in the \( j \)-th column is different from \( t \) (which is again handled by the subexpression \( (T_0 - \{t\}) \)), the expression allows every tile in the \( j \)-th column in the next row. It is easy to see that every string matching the subexpression in the Kleene star has length \( 2n \), and that the expression Ver-even\(_{t,j}\) is one-unambiguous.
Finally, we define the expressions \( \text{Ver-last}_{t,j} \) as follows:

\[
\text{Ver-last}_{t,j} := (T_0^{2n})^* T'^{-1} \\
\left[ (t' T'^{n-j} T'^{-1} (s'_1 + \cdots + s'_\ell)) + ((T' - \{t'\}) T'^{n-j} T'^{n-j}) \right]^* T'^{-1}
\]

If the \( j \)-th tile of the second-last row of the tiling is \( t' \), it matches the leftmost occurrence of \( t' \) in \( \text{Ver-last}_{t,j} \). The expression then ensures that the \( j \)-th tile in the last row is in \( \{s''_1 + \cdots + s''_\ell\} \). If the \( j \)-th tile of the second-last row is different from \( t' \) (which is handled by the subexpression \( (T' - \{t'\}) \)), the expression allows every tile in \( T'' \) in the \( j \)-th position of the last row. It is easy to see that this expression is one-unambiguous.

From the above discussion, it now follows that (\( \diamond \)) holds. Hence, we have shown (a).

We proceed by showing how the maximal number of occurrences of each alphabet symbol can be reduced to three. Notice that we can assume that the given tiling system \( D \) uses pairwise disjoint alphabets \( T_j \) to tile the \( j \)-th column of a tiling. Moreover, we can also assume that \( D \) uses pairwise disjoint alphabets to tile the odd rows and the even rows of the tiling, respectively. Hence, the tiles of \( D \) are partitioned into the pairwise disjoint sets \( T_{j,\text{odd}} \), \( T_{j,\text{even}} \), \( T'_{j,\text{odd}} \), and \( T'_{j,\text{even}} \) for every \( j = 1, \ldots, n \). Here, \( T'_{j,\text{odd}} \) and \( T'_{j,\text{even}} \) are the sets that are used to tile the \( j \)-th column of the second-last and the last row of the tiling, respectively.

When we assume that \( D \) meets these requirements, it is straightforward to verify that, in the above defined regular expressions, every tile of \( T \) occurs at most three times. This concludes the proof of (b).

**Corollary 6.5.** Intersection is \( \text{pspace-complete} \) for one-unambiguous \( \text{RE}^{\leq 3} \) expressions.

A tractable fragment is the following:

**Theorem 6.6.** Intersection is in \( \text{ptime} \) for \( \text{RE}(a, a^+) \).

*Proof.* Suppose we are given \( \text{RE}(a, a^+) \) expressions \( r_1, \ldots, r_n \) in sequence normal form. We describe a \( \text{ptime} \) method to decide non-emptiness of \( L(r_1) \cap \cdots \cap L(r_n) \).

First of all, the intersection can only be non-empty, if all \( r_i \) have the same number \( m \) of factors and, for each \( j \leq n \), the \( j \)-th factor of each \( r_i \) has the same base symbol \( a_j \). That is, each \( r_i \) can be written as \( e_{i,1} \cdots e_{i,n} \) and each \( e_{i,j} \) is of the form \( a_j[k_j^i, l_j^i] \), for some \( k_j^i, l_j^i \geq 1 \).

Let, for each \( j \leq m \), \( p_j := \max\{k_j^i \mid i \leq n\} \) and \( q_j := \min\{l_j^i \mid i \leq n\} \). It is easy to check that \( L(r_1) \cap \cdots \cap L(r_n) \) is non-empty, if and only if, for each \( j \leq m \), \( p_j \leq q_j \). \( \square \)
7 Conclusion

We revisited the inclusion, equivalence and intersection problem for regular expressions that are common in practical DTDs and XML Schemas. Moreover, we showed that for all these problems, the complexities of the inclusion, equivalence and intersection problems carry over to the corresponding problems for DTDs. For inclusion and equivalence, the complexity bounds for regular expressions also carry over to single-type and restrained competition extended DTDs. We left the following complexities open: (i) intersection of \( \text{RE}(a, w^+ \}) \) expressions; and, equivalence for \( \text{RE}(S) \) or any fragment extending \( \text{RE}(a, a^*) \) or \( \text{RE}(a, a?) \).

Acknowledgment

We thank Sebastian Bala for referring us to his work [Bal02], which contains the proof ideas that are needed for Lemma 6.1(c).

References


[BMNS05] G. J. Bex, W. Martens, F. Neven, and T. Schwentick. Expressiveness of XSDs: from practice to theory, there and back again. In


