COMPLEXITY OF DECISION PROBLEMS FOR XML SCHEMAS AND CHAIN REGULAR EXPRESSIONS[†]

WIM MARTENS[‡¶], FRANK NEVEN[§], AND THOMAS SCHWENTICK[‡]

Abstract. We study the complexity of the inclusion, equivalence, and intersection problem of extended CHAin Regular Expressions (eCHAREs). These are 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 "?". Though of a very simple from, the usage of such expressions is widespread as eCHAREs, for instance, constitute a super class of the regular expressions most frequently used in practice in schema languages for XML. In particular, we show 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. These grammars 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. In this respect, we also study two other classes of regular expressions related to XML: deterministic expressions and expressions where the number of occurrences of alphabet symbols is bounded by a constant.

1. Introduction. Although the complexity of the basic decision problems for regular expressions (inclusion, equivalence, and non-emptiness of intersection) have been studied in depth in the seventies [25, 28, 51], the fragment of (extended) CHAin Regular Expressions (eCHAREs) has been largely untreated and is motivated by the much more recent rise of XML Theory [29, 42, 45, 55]. Although our initial motivation to study this particular fragment stems from our interest in schema languages for XML, simple regular expressions like eCHAREs also occur outside the realm of XML. For instance, eCHAREs are a superset of the sequence motifs used in bioinformatics [39] and are used in verification of lossy channel systems [1], where they appear as factors of *simple regular expressions*.[1] The presentation of the paper is therefore split up in two parts. The first part considers the complexity of the basic decision problems for eCHAREs and two other classes. The second part shows how results on decision problems for regular expressions can be lifted to corresponding results on decision problems for XML schema languages.

1.1. Extended Chain Regular Expressions. In brief, an eCHARE is an expression of the form $e_1 \cdots e_n$, where every e_i in turn is a factor of the form $(w_1 + \cdots + w_m)$ — possibly extended with Kleene-star, plus or question mark — and each w_i is a string.[2] Table 1.1 provides a total list of the factors that we allow, with an abbreviated notation for each of them. As an example, $(a^* + b^*)^+ cd(ab + b)^*$ is an eCHARE, whereas, $(a + b)^* + (c + d)^*$ is not. Our interest in eCHAREs is confirmed by a study by Bex, Neven, and Van den Bussche [8] stipulating that the form of most regular expressions occurring in practical XML schema languages like DTDs and XSDs is not

[†]An abstract of a part of this paper already appeared as reference [33] in the International Symposium on Mathematical Foundations of Computer Science, 2004.

[‡]Technical University of Dortmund, Germany. Email addresses: wim.martens@udo.edu (Wim Martens), thomas.schwentick@udo.edu (Thomas Schwentick)

[§]Hasselt University and Transnational University of Limburg, School for Information Technology. Email address: frank.neven@uhasselt.be

[¶]Supported by the North-Rhine Westphalian Academy of Sciences, Humanities and Arts; and the Stiftung Mercator Essen.

[1]A simple regular expression is a disjunction of eCHAREs.

[2]We refer to Section 2 for a detailed definition of extended chain regular expressions.

Factor	Abbr.
a	a
a^*	a^*
a^+	a^+
$a?$	$a?$
w^*	w^*
w^+	w^+
$w?$	$w?$

Factor	Abbr.
$(a_1 + \cdots + a_n)$	$(+a)$
$(a_1 + \cdots + a_n)^*$	$(+a)^*$
$(a_1 + \cdots + a_n)^+$	$(+a)^+$
$(a_1 + \cdots + a_n)?$	$(+a)?$
$(a_1^* + \cdots + a_n^*)$	$(+a^*)$
$(a_1^+ + \cdots + a_n^+)$	$(+a^+)$

Factor	Abbr.
$(w_1 + \cdots + w_n)$	$(+w)$
$(w_1 + \cdots + w_n)^*$	$(+w)^*$
$(w_1 + \cdots + w_n)^+$	$(+w)^+$
$(w_1 + \cdots + w_n)?$	$(+w)?$
$(w_1^* + \cdots + w_n^*)$	$(+w^*)$
$(w_1^+ + \cdots + w_n^+)$	$(+w^+)$

TABLE 1.1

Possible factors in extended chain regular expressions and how they are denoted ($a, a_i \in \Sigma$, $w, w_i \in \Sigma^+$).

arbitrary but adheres to the structure of eCHAREs. In a follow-up examination of the investigated corpus [7], it was observed that the number of occurrences of the same symbol in a regular expressions is rather low, most of the time even equal to one. We therefore also consider the class $\mathrm{RE}^{\leq k}$ of arbitrary regular expressions where every symbol can occur at most k times. A third class we consider, are the one-unambiguous or deterministic regular expressions [10]. They are discussed in more detail in Section 1.2.

RE-fragment	INCLUSION	EQUIVALENCE	INTERSECTION
a, a^+	in PTIME (DFA!)	in PTIME	in PTIME (3.12)
a, a^*	CONP (3.1)	in PTIME (3.6)	NP (3.7)
$a, a?$	CONP (3.1)	in PTIME (3.6)	NP (3.7)
$a, (+a^+)$	CONP (3.1)	in CONP	NP (3.7)
$a^+, (+a)$	CONP (3.1)	in CONP	NP (3.7)
a, w^+	CONP (3.1)	in CONP	in NP (3.7)
$\mathit{all} - \{(+a)^*, (+w)^*, (+a)^+, (+w)^+\}$	CONP (3.1)	in CONP	NP (3.7)
$a, (+a)^*$	PSPACE (3.1)	in PSPACE	NP (3.7)
$a, (+a)^+$	PSPACE (3.1)	in PSPACE	NP (3.7)
$\mathit{all} - \{(+w)^*, (+w)^+\}$	PSPACE (3.1)	in PSPACE	NP (3.7)
$a, (+w)^*$	PSPACE (3.1)	in PSPACE	PSPACE ([4])
$a, (+w)^+$	PSPACE (3.1)	in PSPACE	PSPACE ([4])
all	PSPACE (3.1)	in PSPACE	PSPACE ([4])
$\mathrm{RE}^{\leq k}$ $(k \geq 3)$	in PTIME (3.2)	in PTIME	PSPACE (3.10)
one-unambiguous	in PTIME	in PTIME	PSPACE (3.10)

TABLE 1.2

Summary of our results. Unless specified otherwise, all complexities are completeness results. The theorem numbers are given in brackets. The mentioned complexities for INCLUSION *and* EQUIVALENCE *also hold for DTDs, single-type EDTDs, and restrained competition EDTDs with the respective regular expressions (Theorem 4.10). The complexities for* INTERSECTION *also hold for DTDs with the respective regular expressions (Theorem 4.14).*

We consider the following problems for regular expressions:

- INCLUSION: Given two expressions r and r', is every string in r also defined by r'?
- EQUIVALENCE: Given two expressions r and r', do r and r' define the same set of strings?

- INTERSECTION: Given $r_1, \dots, r_n$, do they define a common string?

Recall that these three decision problems are PSPACE-complete for the class of all regular expressions [28, 51]. We briefly discuss our results. They are summarized in Table 1.2.

- 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 $a, a_1, \dots, a_n$ 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 for $\mathrm{RE}^{\leq k}$).
- 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 the eCHARES. 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 NP-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 PSPACE-complete, whereas INTERSECTION problem is NP-complete. Indeed, INTERSECTION remains in NP 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 PSPACE-hard for $\mathrm{RE}^{\leq 3}$ and for deterministic (or *one-unambiguous*) regular expressions [12], whereas their inclusion problem is in PTIME. The only tractable fragment we obtain is when each factor is restricted to a or a^+.

We denote subclasses of extended chain regular expressions by $\mathrm{RE}(X)$, where X is a list of the allowed factors. For example, we write $\mathrm{RE}((+a)^*, w?)$ for the set of regular expressions $e_1 \cdots e_n$ where every e_i is either *(i)* $(a_1 + \cdots + a_m)^*$ for some $a_1, \dots, a_m \in \Sigma$ and $m \geq 1$, or *(ii)* $w?$ for some $w \in \Sigma^+$. As mentioned above, the complexity of the basic decision problems for regular expressions have been studied in depth [25, 28, 51]. From these, the most related result is the coNP-completeness of EQUIVALENCE and INCLUSION of bounded languages [25]. A language L is *bounded* if there are strings $v_1, \dots, v_n$ 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 $\mathrm{RE}(w^*)$. More recently, INCLUSION for two fragments of extended chain regular expressions have been shown to be tractable: INCLUSION for $\mathrm{RE}(a?, (+a)^*)$ [1] and $\mathrm{RE}(a, \Sigma, \Sigma^*)$ [37, 36]. This last result should be contrasted with the PSPACE-completeness of INCLUSION for $\mathrm{RE}(a, (+a), (+a)^*)$, or even $\mathrm{RE}(a, (+a)^*)$. Further, Bala investigated INTERSECTION for regular expressions of limited star height [4]. He showed that it is PSPACE-complete to decide whether INTERSECTION for $\mathrm{RE}((+w)^*)$ expressions contains a non-empty string. We show how to adapt Bala's proof to obtain PSPACE-hardness of INTERSECTION for $\mathrm{RE}(a, (+w)^*)$

or $\mathrm{RE}(a, (+w)^+)$ expressions. The study of eCHAREs has been extended to include numerical occurrence constraints in [20] while [21] obtained a fragment of eCHAREs admitting a polynomial time algorithm for inclusion. In [7], algorithms have been proposed to learn general $\mathrm{RE}^{\leq 1}$-expressions and expressions which are both $\mathrm{RE}^{\leq 1}$ and eCHAREs.

1.2. XML Theory: Schema Optimization. XML is an abbreviation for eXtensible Markup Language and has become the standard data exchange format for the World Wide Web [2]. As XML documents can be adequately abstracted by labeled trees (with or without links) many tools and techniques from formal language theory have been revisited in this context. More specifically, a renewed interest has grown in the specification formalism of regular expressions. For instance they are included in navigational queries expressed by caterpillar expressions [13], $\mathcal{X}_{\mathrm{reg}}^{\mathrm{CPath}}$, $\mathcal{X}_{\mathrm{reg}}$, and XPath with transitive closure [35, 52], and regular path queries [14]. Therefore, lower bounds for optimization problems for eCHAREs readily imply lower bound for optimization problems for navigational queries.

Another application of regular expressions surfaces in the description of schemas for XML documents. 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., [5, 17, 27, 31, 43, 56]). Furthermore, for typechecking or type inference of XML transformations [24, 32, 38, 44], schema information is even crucial. The INCLUSION, EQUIVALENCE, and INTERSECTION problems for schemas are among the basic building blocks for many of the algorithms for the above mentioned problems. They are defined similarly as for regular expressions:

- 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.

Although many XML schema languages have been proposed, Document Type Definitions (DTDs) [9], XML Schema Definitions (XSDs) [50], and Relax NG [40] are the most prevalent ones. In particular, DTDs correspond to context-free grammars with regular expressions (REs) at right-hand sides, while Relax NG is abstracted by extended DTDs (EDTDs) [44] or equivalently, unranked tree automata [11], defining the regular unranked tree languages. While XML Schema is usually abstracted by unranked tree automata as well, recent results indicate that XSDs correspond to a strict subclass of the regular tree languages and are much closer to DTDs than to tree automata [34]. In fact, they can be abstracted by single-type EDTDs. We also consider *restrained competition* EDTDs [41, 34], 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 [34, 6]. 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).

As grammars and tree automata have already been studied in depth for many decades, it is not surprising that the complexity of some of the above mentioned deci-

sion 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 [28, 51]. For tree automata, they are well-known to be EXPTIME-complete [47, 48].

We discuss the determinism constraint imposed on regular expressions: the XML specification requires DTD content models to be *deterministic* because of compatibility with SGML (Section 3.2.1 of [9]). In XML Schema, this determinism constraint is referred to as the *unique particle attribution* constraint (Section 3.8.6 of [50]). Brüggemann-Klein and Wood[12] 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 follows that INCLUSION and EQUIVALENCE for such regular expressions, and hence also for practical DTDs and XSDs, are in PTIME. In contrast, we show in Theorem 3.10 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 eCHAREs 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 [53] and [30, 49]). 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 [8, 16] that found a number of non-deterministic content models in actual DTDs. In practice, XSDs allow numerical occurrence constraints in their regular expressions, which complicates the definition of determinism even more [26, 19]. In fact, Clarke and Murata already abandoned the notion in their Relax NG specification [54], the most serious competitor for XML Schema.

From our results, it becomes clear that, if one would choose to abandon the one-unambiguity constraint in DTD or XML Schema, the complexity of INCLUSION and EQUIVALENCE would significantly increase, even for regular expressions that have severe syntactical restrictions.

In the second part of this paper, we present a finer analysis of the complexity of the basic decision problems for schema languages for XML. Clearly, complexity lower bounds for INCLUSION, EQUIVALENCE, or INTERSECTION for a class of regular expressions $\mathcal{R}$ imply lower bounds for the corresponding decision problems for DTDs, single-type EDTDs, and restrained competition EDTDs with right-hand sides in $\mathcal{R}$. Probably one of the most important results of this paper is 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.

In this respect, the second part of the paper can be used to revive the interest in studying the complexity of INCLUSION and EQUIVALENCE for regular expressions, as most such results will carry over to DTDs and single-type EDTDs. In particular, all the results in this paper on subclasses of regular expressions carry over to the considered XML schema languages.

Organization. In Section 2, we introduce the necessary definitions and show some preliminary lemmas. In Section 3, we consider inclusion, equivalence, and non-emptiness of intersection of extended chain regular expressions, respectively. Here, we also treat $\mathrm{RE}^{\leq k}$ and one-unambiguous regular expressions. In Section 4, we relate decision problems for XML schemas to the corresponding decision problems on

regular expressions. In Section 5, we present concluding remarks.

2. Preliminaries. For the rest of the paper, Σ always denotes a finite alphabet. A Σ-*symbol* (or simply symbol) is an element of Σ, and a Σ-*string* (or simply string) is a finite sequence $w = a_1 \cdots a_n$ of Σ-symbols. We define the length of w, denoted by $|w|$, to be n. We denote the empty string by ε. The set of *positions of* w is $\{1, \ldots, n\}$ 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 often also denote the concatenation of w_1 and w_2 by $w_1 w_2$. The set of all Σ-strings is denoted by Σ^*. A *string language* is a subset of Σ^*, for some Σ. 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. If $w = a_1 \cdots w_n$ is a string and $i \leq j+1$, we write $w[i,j]$ for the substring $a_i \cdots a_j$. Note that $w[i+1,i]$ is always the empty string.

2.1. Regular Expressions. The set of *regular expressions* over Σ, denoted by RE, is defined in the usual way:

- $\emptyset$, ε, and every Σ-symbol is a regular expression; and
- when r and s are regular expressions, then $r \cdot s$, $r + s$, and r^* are also regular expressions.

Brackets can be added to improve readability and to disambiguate expressions.

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

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

The *size* of a regular expression r over Σ, denoted by $|r|$, is the number of $\Sigma \uplus \{+, *\}$-symbols occurring in r. By $r?$ and r^+, we abbreviate the expressions $r + \varepsilon$ and rr^*, respectively.

In many of our proofs, we reason about 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 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 has to be 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 $r_2 \in m(i,j)$;
(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 $r \in m(k_\ell + 1, k_{\ell+1})$, for all ℓ, $1 \leq \ell < t$.

Furthermore, m has to be 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 $r' \in m(i,j)$ and write $w[i,j] \models_m r'$. Often, we leave m implicit whenever this cannot give rise to confusion. We then say that $a_i \cdots a_j$ *matches* r' or write $w[i,j] \models r'$. It should be noted that m is uniquely determined by the values $m(i,i)$, for all positions i. Therefore, we sometimes write $m(i)$ for the unique symbol from σ in $m(i,i)$.

The eCHAREs are formally 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. An* extended chain regular expression (eCHARE) *is $\emptyset$, ε, or a concatenation of factors.*

For example, the regular expression $((abc)^* + b^*)(a + b)?(ab)^+(ac + b)^*$ is an extended chain regular expression. (We give an explanation below.) The expression $(a + b) + (a^*b^*)$, however, is not, due to the nested disjunction and the nesting of Kleene star with concatenation.

We introduce a uniform syntax to denote subclasses of extended 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 non-empty string (denoted by w) and whether it is extended by $*$, $+$, or $?$. Furthermore, we distinguish between factors with one disjunct or with arbitrarily many disjuncts: the latter is denoted by $(+\cdots)$. Finally, factors can again be extended by $*$, $+$, or $?$. A list of possible factors, together with their abbreviated notation, is displayed in Table 1.1. This table only shows factors which give rise to extended chain regular expressions with different expressive power.

The regular expression $((abc)^* + b^*)(a + b)?(ab)^+(ac + b)^*$ mentioned above has, in the notation of Table 1.1, factors of the form $(+w^*)$, $(+a)?$, w^+, and $(+w)^*$, from left to right.

In the remainder of the paper, we restrict ourselves to the factors listed in Table 1.1. Notice that other factors, such as, e.g., $(w_1^* + \cdots + w_n^*)^*$ or $(w_1? + \cdots + w_n?)^*$ are superfluous because they can be equivalently rewritten as $(w_1 + \cdots + w_n)^*$. Similar rewritings are possible for other factors not listed in Table 1.1.

We denote subclasses of extended chain regular expressions by RE(X), where X is a list of the allowed factors. For example, we write RE($(+a)^*, w?$) for the set of regular expressions $e_1 \cdots e_n$ where every e_i is either *(i)* $(a_1 + \cdots + a_m)^*$ for some $a_1, \ldots, a_m \in \Sigma$ and $m \geq 1$, or *(ii)* $w?$ for some $w \in \Sigma^+$.

If $A = \{a_1, \ldots, a_n\}$ is a set of symbols, we often denote $(a_1 + \ldots + a_n)$ simply by A. We denote the class of all extended chain regular expressions by RE(all).

2.1.1. One-Unambiguous Regular Expressions. We recall the notion of one-unambiguous regular expressions [12]. 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 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, then $(a_1a_3b_4)^\# = aab$.

Definition 2.2. *A regular expression r is* one-unambiguous *if for all strings u, v, w and symbols x, y, the conditions uxv, $uyw \in L(mark(r))$ and $x \neq y$ imply that $x^\# \neq y^\#$.* The intuition behind one-unambiguous regular expressions is that, when reading a string from left to right, there is always at most one Σ-symbol in the regular expression that matches the current symbol (while being consistent with the subexpressions that match the prefix of the string that is already read). One-unambiguous regular expressions are therefore also referred to as *deterministic regular expressions* in the literature. For instance, $r_1 = (a + b)(a + b)^*$ is a one-unambiguous

regular expression, while $r_2 = (a + b)^*(a + b)$ is not. Intuitively, r_2 is not one-unambiguous as when reading a string starting with an a, there is no way of knowing whether to match the first or the second a in r_2. Indeed, $\text{mark}(r_2) = (a_1+b_2)^*(a_3+b_4)$, $a_1a_3 \in \text{mark}(r_2)$ 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.

2.2. Decision Problems. The following three problems are fundamental to this paper.

DEFINITION 2.3. *Let $\mathcal{R}$ be a class of regular expressions.*

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

Notice that if $\mathcal{R}$ is the class of arbitrary regular expressions all these problems are PSPACE-complete [28, 51]. Given two regular expressions r and r', we also say that r *is included in* r' (resp., *is equivalent to*) if $L(r) \subseteq L(r')$ (resp., $L(r) = L(r')$).

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, Σ 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 $\delta(q, a)$ contains at most one element 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 $i \in \mathbf{N} - \{0\}$. The pair (w, i) stands for the string w^i. The *size* of a pair (w, i) is $|w| + \lceil \log i \rceil$. The size of a compressed string $v = (w_1, i_i) \cdots (w_n, i_n)$ is the sum of the sizes of $(w_1, i_1), \ldots, (w_n, i_n)$. By string$(v)$, we denote the *decompressed* string corresponding to v, which is the string $w_1^{i_1} \cdots w_n^{i_n}$. Notice that string$(v)$ can be exponentially larger than v.

The following lemma shows that we can decide in polynomial time whether a compressed string is accepted by an NFA. We use it later in the article as a tool to obtain complexity upper bounds.

LEMMA 2.4. *Given a compressed string v and an NFA A, we can test whether 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 [51], which shows that the equivalence problem for regular expressions over a unary alphabet is 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 $Q_A = \{1, \ldots, k\}$. The goal is to test in polynomial time whether $\delta^*(I_A, \text{string}(v)) \cap F_A = \emptyset$. It is sufficient to compute, for each j, the transition relation R_j, i.e., all pairs (p, q) of states of A such that $q \in \delta(p, \text{string}(w_j, i_j))$. To this end, we first compute the transition relation of w_j and get R_j by $\lceil \log i_j \rceil$ iterated squarings (sometimes also called successive squaring [46]). ∎

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 $\lambda : \{1, \ldots, m\} \times \{1, \ldots, n\} \to T$, for some $m \in \mathbf{N}$, 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 $j = 1, \dots, n-1$, $(\lambda(i,j), \lambda(i,j+1)) \in H$, and for every $i = 1, \dots, m-1$ and $j = 1, \dots, n$, $(\lambda(i,j), \lambda(i+1,j)) \in V$.

With every tiling system we associate a *tiling game* as follows: the game consists of two players (CONSTRUCTOR and SPOILER). The game is played on an $\mathbb{N} \times n$ board. Each player places tiles in turn. While CONSTRUCTOR tries to construct a corridor tiling, SPOILER tries to prevent it. CONSTRUCTOR wins if SPOILER makes an illegal move (with respect to H or V), or when a correct corridor tiling is completed. 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 in the corresponding tiling game?

The following theorem is due to Chlebus [15].

THEOREM 2.5.

1. CORRIDOR TILING *is* PSPACE-*complete.*
2. TWO-PLAYER CORRIDOR TILING *is* EXPTIME-*complete.*

3. Decision Problems for Chain Regular Expressions. We turn to the complexity of INCLUSION, EQUIVALENCE, and INTERSECTION for the chain regular expressions themselves.

3.1. Inclusion. 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 $\mathrm{RE}(a?, (+a)^*)$ can be solved in linear time. Whether $p \subseteq p_1 + \cdots + p_k$ for $p, p_1, \dots, p_k$ from $\mathrm{RE}(a?, (+a)^*)$ can be checked in quadratic time [1].
- In [37] it is stated that INCLUSION for $\mathrm{RE}(a, \Sigma, \Sigma^*)$ is in PTIME. In this fragment, only disjunctions over the full alphabet are allowed. A proof can be found in [36].

Some of the fragments we defined are so small that one expects their containment problem to be tractable. Therefore, it came as a surprise to us that even for $\mathrm{RE}(a, a?)$ and $\mathrm{RE}(a, a^*)$ the inclusion problem is already CONP-complete, especially when comparing to the linear time result for, e.g., $RE(a?, (+a)^*)$. That INCLUSION for $\mathrm{RE}(a?, (+a)^*)$ is in linear time is quite easy to see: given two expressions, r_1 and r_2, one can use a greedy algorithm that reads the expressions from left to right and tries to match as much as possible of r_1 into r_2. If everything of r_1 is matched, we are done because the remainder of r_2 always matches the empty string. The main difference when adding the a-factor is that the remainder of the expression r_2 can now require that r_1 still needs to produce *at least* some symbols, hence, this greedy left-to-right approach does no longer work.

Maybe even more surprising is that $\mathrm{RE}(a, (+a)^*)$ and $\mathrm{RE}(a, (+a)^+)$ already yield the maximum complexity: PSPACE-completeness. The PSPACE-hardness of inclusion of $\mathrm{RE}(a, (+a)^*)$ expressions highly contrasts with the PTIME inclusion for $\mathrm{RE}(a, \Sigma, \Sigma^*)$ mentioned above.

Our results, together with corresponding upper bounds are summarized in Theorem 3.1. Let $\mathrm{RE}(\mathrm{all} - \{(+a)^*, (+w)^*, (+a)^+, (+w)^+\})$ denote the fragment of RE(all) where no factors of the form $(a_1 + \cdots + a_n)^*$, $(w_1 + \cdots + w_n)^*$, $(a_1 + \cdots + a_n)^+$ or $(w_1 + \cdots + w_n)^+$ are allowed for $n \geq 2$.

THEOREM 3.1.
(a) INCLUSION *is* coNP*-hard for*
  (1) $RE(a, a^*)$,
  (2) $RE(a, a?)$,
  (3) $RE(a, w^+)$,
  (4) $RE(a, (+a^+))$, *and*
  (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* $RE(all - \{(+a)^*, (+w)^*, (+a)^+, (+w)^+\})$; *and*
(d) INCLUSION *is in* PSPACE *for* $RE(all)$.

Theorem 3.1 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 $\mathrm{RE}(a, a^+)$. Another case is covered by the following theorem.

THEOREM 3.2. INCLUSION *for* $RE^{\leq k}$ *is in* PTIME *when* k *is fixed.*

Proof. Let r be an $\mathrm{RE}^{\leq k}$ expression. Let $A_r = (Q, \Sigma, \delta, I, F)$ be the Glushkov automaton for r [22] (see also [12]). 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 $\mathcal{O}(n^k)$, 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 over ninety percent of the regular expressions DTDs or XML Schemas, k is equal to one [7].

In the rest of this section, we prove Theorem 3.1. We start by showing Theorem 3.1(a), i.e., that INCLUSION is coNP-hard for
(1) $\mathrm{RE}(a, a^*)$,
(2) $\mathrm{RE}(a, a?)$,
(3) $\mathrm{RE}(a, (+a^+))$,
(4) $\mathrm{RE}(a, w^+)$; and
(5) $\mathrm{RE}(a^+, (+a))$.

In all five cases, we construct a LOGSPACE reduction from VALIDITY of propositional formulas in disjunctive normal form with exactly three literals per clause (3DNF). The VALIDITY problem asks, given a propositional formula Φ in 3DNF with variables $\{x_1, \ldots, x_n\}$, whether Φ is true under all truth assignments for $\{x_1, \ldots, x_n\}$. The VALIDITY problem for 3DNF formulas is known to be coNP-complete [18]. We note that, for the cases (1)–(3), we even show that INCLUSION is already coNP-hard when the expressions use a fixed-sized 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 coNP-hard [36]. We present a robust generalization of the technique and use it to show coNP-hardness of all five fragments.

We now proceed with the proof. Thereto, let $\Phi = C_1 \vee \cdots \vee C_k$ 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

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

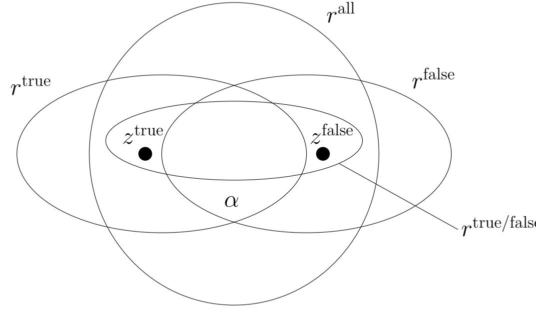


FIG. 3.1. *Inclusion structure of regular expressions used in* coNP*-hardness of* INCLUSION.

More specifically, we encode truth assignments for Φ 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 Φ true.

To define R_1 and R_2 we use auxiliary regular expressions W and N, a string u and expressions $F(C)$ associated with clauses C:

- The strings of $L(W)$ will 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.
- For each clause C and every $w \in L(W)$ it will hold that $w \models F(C)$ if and only if $A_w \models C$.
- The string u matches every expression $F(C)$.
- Finally, N can match the concatenation of up to k occurrences of the string u.

Then we define

$$\begin{aligned} R_1 &= u^k W u^k, \\ R_2 &= N F(C_1) \cdots F(C_k) N. \end{aligned}$$

Intuitively, $L(R_1) \subseteq L(R_2)$ now holds if and only if every truth assignment (matched by the middle part of R_1) makes at least one disjunct true (i.e., matches at least one $F(C_i)$).

For each of the cases (1)–(5), we construct the auxiliary expressions with the help of five basic regular expressions r^{true}, r^{false}, $r^{\text{true/false}}$, r^{all}, and α, which must adhere to the inclusion structure graphically represented in Figure 3.1 and formally stated by the following properties (INC1)–(INC5).

$$\alpha \in L(r^{\text{false}}) \cap L(r^{\text{true}}) \tag{INC1}$$

$$L(r^{\text{true/false}}) \subseteq L(r^{\text{false}}) \cup L(r^{\text{true}}) \tag{INC2}$$

$$L(r^{\text{true/false}}) \cup \{\alpha\} \subseteq L(r^{\text{all}}) \tag{INC3}$$

$$z^{\text{true}} \in L(r^{\text{true/false}}) - L(r^{\text{false}}) \tag{INC4}$$

$$z^{\text{false}} \in L(r^{\text{true/false}}) - L(r^{\text{true}}) \tag{INC5}$$

Intuitively, the two dots in Figure 3.1 are strings z^{true} and z^{false}, which represent the truth values *true* and *false*, respectively. The expressions r^{true} and r^{false} are used to match z^{true} and z^{false} in $F(C)$, respectively. The expression $r^{\text{true/false}}$ is used in

	(1) RE(a, a^*)	(2) RE$(a, a?)$	(3) RE(a, w^+)
α	a	a	$aaaa$
r^{true}	$aa^*b^*a^*$	$aa?$	$a^+(aa)^+$
r^{false}	b^*a^*	$a?$	$(aa)^+$
$r^{\text{true/false}}$	$a^*b^*a^*$	$a?a?$	aa^+
r^{all}	$a^*b^*a^*$	$a?a?$	a^+
z^{true}	ab	aa	aaa
z^{false}	ba	ε	aa
W	$\#r^{\text{true/false}}\$\cdots$ $\cdots\$r^{\text{true/false}}\#$	$\#r^{\text{true/false}}\$\cdots$ $\cdots\$r^{\text{true/false}}\#$	$\#r^{\text{true/false}}\$\cdots$ $\cdots\$r^{\text{true/false}}\#$
N	$(\#^*a^*\$^*\cdots\$^*a^*\#^*)^k$	$(\#?a?\$?\cdots\$?a?\#?)^k$	$(\#aaaa\$\cdots\$aaaa\#)^+$
u	$\#\alpha\$\cdots\$\alpha\#$	$\#\alpha\$\cdots\$\alpha\#$	$\#\alpha\$\cdots\$\alpha\#$

TABLE 3.1
Definition of basic and auxiliary expressions for cases (1)–(3). In the definition of W, N and u the repetition indicated by $\cdots$ is always n-fold and does not include $\#$.

W to generate both, *true* and *false*. Finally, α is used in u and r^{all} is used for the definition of $F(C)$.

The definition of the basics expressions and of W, N and u for cases (1)-(3) is given in Table 3.1. We will consider cases (4) and (5) later.

It is straighforward to verify that the conditions (INC1)–(INC5) are fulfilled for each of the fragments. Note that the expressions use the fixed alphabet $\{a, b, \$, \#\}$.

The association between strings and truth assignments is now straightforward.

- For $w = \#w_1\$\cdots\$w_n\# \in L(W)$ let A_w be defined as follows:

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

- For a truth assignment A, let $w_A = \#w_1\$\cdots\$w_n\#$, where, for each $j = 1, \ldots, n$,

$$w_j = \begin{cases} z^{\text{true}} & \text{if } A(x_j) = \text{true} \\ z^{\text{false}} & \text{otherwise.} \end{cases}$$

- Finally, for each clause C, we define $F(C) = \#e_1\$\cdots\$e_n\#$, where for each $j = 1, \ldots, n$,

$$e_j := \begin{cases} r^{\text{false}}, & \text{if } x_j \text{ occurs negated in } C, \\ r^{\text{true}}, & \text{if } x_j \text{ occurs positively in } C, \text{ and} \\ r^{\text{all}}, & \text{otherwise.} \end{cases}$$

CLAIM 3.3. *If $A_w \models C$ then $w \in L(F(C))$. If $w_A \in L(F(C))$ then $A \models C$.*

Proof of Claim 3.3. Let $w = \#w_1\$\cdots\$w_n\# \in W$ be a string with $A_w \models C$ and let $F(C) = \#e_1\$\cdots\$e_n\#$ be as defined above. We show that $w \in L(F(C))$. To this end, let $j \leq n$. There are three cases to consider:

1. If x_j does not occur in C then $e_j = r^{\text{all}}$. 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 in C, then $e_j = r^{\text{true}}$. As $A_w \models C$, $A_w(x_j) = \text{true}$ and, by definition of A_w, we get $w_j \in L(r^{\text{true}}) = L(e_j)$.
3. If x_j occurs negatively in C, then $e_j = r^{\text{false}}$. As $A_w \models C_i$, $A_w(x_j) = \text{false}$ and thus $w_j \notin L(r^{\text{true}})$ by definition of A_w. Thus, $w_j \in L(r^{\text{true/false}}) - L(r^{\text{true}})$ and thus in $L(e_j) = L(r^{\text{false}})$ by condition (INC2).

Therefore, for each $j = 1, \ldots, n$, $w_j \in L(e_j)$ and thus $w \in L(F(C))$.

We show the other statement by contraposition. Thereto, let A be a truth assignment such that does not make clause C true. We show that $w_A \notin L(F(C))$. There are two cases:

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

In the proof of $(*)$ we will make use of the following properties.

Claim 3.4.

(a) $u^i \in L(N)$ *for every* $i = 1, \ldots, k$.
(b) $u \in L(F(C_i))$ *for every* $i = 1, \ldots, k$.
(c) If $u^k w u^k \in L(R_1) \cap L(R_2)$*, then* w *matches some* F_i.

Proof of Claim 3.4. (a) can be easily checked and (b) follows immediately from conditions (INC1), (INC3), and the definition of F. In the following, we abbreviate $F(C_i)$ by F_i. To show (c), 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 $\#y_1\#\#y_2\# \cdots \#y_\ell\#$, where $y_1, \ldots, y_\ell$ are non-empty strings over $\{a, \$\}$. Hence, as $u^k w u^k \in L(R_2)$ and as none of the strings $y, y_1, \ldots, y_\ell$ contain the symbol "#", w either matches some F_i or w matches a subexpression 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 $u^k w$ 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 $w u^k$ matches the right occurrence of the expression N in R_2. So, m must match w onto some F_i.
- 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 expresion N in R_2. Observe that every string that matches $F_1 \cdots F_k N$ is of the form $\#y''_1\#\#y''_2\# \cdots \#y''_{\ell'}\#$, 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 $w u^k$ onto the right occurrence of the expression N in R_2. So, m must match w onto some F_i. ∎

Now we can complete the proof of $(*)$ for fragments (1)–(3).

Proof of $(*)$. ($\Rightarrow$) Assume that $L(R_1) \subseteq L(R_2)$. Let A be an arbitrary truth assignment for Φ and let $v = u^k w_A u^k$. As $v \in L(R_1)$ and $L(R_1) \subseteq L(R_2)$, also $v \in L(R_2)$. By Claim 3.4 (c), it follows that w_A matches some $F(C)$. Hence, by

	(4) RE($+a^+$)	(5) RE($a^+, (+a)$)
α	a	a
r_j^{true}	$(a^+ + b_j^+)$	$(a + b_j)$
r_j^{false}	$(a^+ + c_j^+)$	$(a + c_j)$
$r_j^{\text{true/false}}$	$(b_j^+ + c_j^+)$	$(b_j + c_j)$
r_j^{all}	$(a^+ + b_j^+ + c_j^+)$	$(a + b_j + c_j)$
z_j^{true}	b_j	b_j
z_j^{false}	c_j	c_j
W	$r_1^{\text{true/false}} \cdots r_n^{\text{true/false}}$	$r_1^{\text{true/false}} \cdots r_n^{\text{true/false}}$
N	a^+	a^+
u	α^n	α^n

Table 3.2
Definition of basic and auxiliary expressions for cases (4) and (5).

Claim 3.3, $A \models C$ and, consequently, $A \models \Phi$. As A is an arbitrary truth assignment, we have that Φ is a valid propositional formula.

($\Leftarrow$) Suppose that Φ is valid. Let v be an arbitrary string in $L(R_1)$. By definition of R_1, v is of the form $v = u^k w u^k$. As Φ is valid, there is a clause C of Φ that becomes true under A_w. Due to Claim 3.3, $w \in L(F(C))$. Furthermore, as $u \models F(C_j)$ for every $j = 1, \ldots, k$ by Claim 3.4 (b), and as $L(N)$ contains u^ℓ for every $\ell = 1, \ldots, k$ by Claim 3.4 (a), we get that $v = u^k w u^k \in L(R_2)$. As v is an arbitrary string in $L(R_1)$, it follows $L(R_1) \subseteq L(R_2)$. $\square$

This completes the proof of Theorem 3.1(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^{true}, r^{false}, $r^{\text{true/false}}$, and r^{all}, we will now have expressions r_j^{true}, r_j^{false}, $r_j^{\text{true/false}}$, and r_j^{all} for every $j = 1, \ldots, n$. We will require that these expressions fulfill the properties (INC1)–(INC5), for each j, where $r^{\text{true}}, r^{\text{false}}, r^{\text{true/false}}, r^{\text{all}}$ are replaced by the expressions r_j^{true}, r_j^{false}, $r_j^{\text{true/false}}$, and r_j^{all}, respectively. The definitions of the basic and auxiliary expressions are given in Table 3.2.

The association between strings and truth assignments is now defined as follows.

- For $w = w_1 \cdots w_n \in L(W)$, where for every $j = 1, \ldots, n$, $w_j \in r_j^{\text{true/false}}$, let A_w be defined as follows:

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

- For a truth assignment A, let $w_A = w_1 \cdots w_n$, where, for each $j = 1, \ldots, n$,

$$w_j = \begin{cases} z_j^{\text{true}} & \text{if } A(x_j) = \text{true and} \\ z_j^{\text{false}} & \text{otherwise.} \end{cases}$$

- Finally, for each clause C, we define $F(C) = e_1 \cdots e_n$, where for each $j =$

$1, \ldots, n$,

$$e_j := \begin{cases} r_j^{\text{false}}, & \text{if } x_j \text{ occurs negated in } C, \\ r_j^{\text{true}}, & \text{if } x_j \text{ occurs positively in } C, \text{ and} \\ r_j^{\text{all}}, & \text{otherwise.} \end{cases}$$

In order to prove $(*)$ for fragments (4) and (5), we have to show that Claims 3.3 and 3.4 hold. For Claim 3.3 this can be done similarly as before, and for Claim 3.4 (a) and (b) it is again easy to see. We complete the proof of Theorem 3.1 by showing Claim 3.4 (c).

To this end, 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. Let m be a match between $u^k w u^k$ and R_2. For every $j = 1, \ldots, n$, let Σ_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 Σ_j^+. Moreover, no strings in $L(N)$ contain symbols from Σ_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 $y'_1 \ldots y'_n$, where each y'_j is a string in $(\Sigma_j \cup \{a\})^+$. As m can only match symbols in Σ_j onto subexpressions with symbols in Σ_j, m matches w onto some F_i.

This concludes the proof of Theorem 3.1(a). ◻

We prove Theorem 3.1(b) by a reduction from Corridor Tiling.

Theorem 3.1(b). inclusion *is* pspace-*hard for*

(1) $RE(a, (+a)^+)$*; and*

(2) $RE(a, (+a)^*)$*.*

Proof. We first show that inclusion is pspace-hard for $\text{RE}(a, (+a)^+)$ and we consider the case of $\text{RE}(a, (+a)^*)$ 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

$$L(R_1) \subseteq L(R_2) \text{ if and only if there exists no correct corridor tiling for } D.$$

We will use symbols from $\Sigma = T \times \{1, \ldots, n\}$, where, for each i, $\Sigma_i = \{[t, i] \mid t \in T\}$ will be used to tile the i-th column. For ease of exposition, we denote $\Sigma \cup \{\$\}$ by $\Sigma_\$$ and $\Sigma \cup \{\#, \$\}$ by $\Sigma_{\#,\$}$. We encode candidates for a correct tiling by strings in which the rows are separated by the symbol \$, that is, by strings of the form

$$\$\bar{b}\$\Sigma^+\$\Sigma^+\$ \cdots \$\Sigma^+\$\bar{t}\$. \qquad (\dagger)$$

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

- $\Sigma_\$^+[t, i][t', j]\Sigma_\$^+$, 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;
- $\Sigma_\$^+\$[t, i]\Sigma_\$^+$ for every $i \neq 1$ and $[t, i] \in \Sigma_i$, and $\Sigma_\$^+[t, i]\$\Sigma_\$^+$ 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 $\Sigma_1 \cdots \Sigma_n$.

- $\Sigma_{\$}^{+}[t,i][t',i+1]'\Sigma_{\$}^{+}$, for every $(t,t') \notin H$, and $i = 1,\dots,n-1$. These expressions detect all violations of horizontal constraints.
- $\Sigma_{\$}^{+}[t,i]\Sigma^{+}\$\Sigma^{+}[t',i]\Sigma_{\$}^{+}$, for every $(t,t') \notin \Sigma$ and for every $i = 1,\dots,n$. These expressions detect all violations of vertical constraints.

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

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

$$\begin{array}{rcl} R_1 & = & \overbrace{\#e\#e\#\cdots\#e\#}^{k \text{ times } e}\$\bar{b}\$\Sigma_{\$}^{+}\$\bar{t}\$\overbrace{\#e\#e\#\cdots\#e\#}^{k \text{ times } e} \text{ and} \\ R_2 & = & \Sigma_{\#,\$}^{+}\#e_1\#e_2\#\cdots\#e_k\#\Sigma_{\#,\$}^{+}. \end{array}$$

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

We first show the implication from left to right. Thereto, assume $L(R_1) \subseteq L(R_2)$. Let uwu' be an arbitrary string in $L(R_1)$ such that $w \in \$\bar{b}\$\Sigma_{\$}^{+}\$\bar{t}\$$ and $u,u' \in L(\#e\#e\#\cdots\#e\#)$. Hence, $uwu' \in L(R_2)$. Let m be a match such that $uwu' \models_m R_2$. Notice that uwu' contains $2k+2$ times the symbol "#". However, as uwu' starts and ends with the symbol "#", m matches the first and the last "#" of uwu' onto the $\Sigma_{\#,\$}^{+}$ sub-expressions of R_2. Thus $vwv' \models_m \#e_1\#e_2\#\cdots\#e_k\#$, for some suffix v of u and prefix v' of u'. In particular, $w \models_m e_i$, for some i. So, w does not encode a correct tiling. As the $\$\bar{b}\$\Sigma_{\$}^{+}\$\bar{t}\$$ defines all candidate tilings, the system D has no solution.

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

The PSPACE-hardness proof for $\mathrm{RE}(a,(+a)^{*})$ is completely analogous, except that every "Σ^{+}" has to be replaced by a "Σ^{*}" and that

$$R_2 = \#\Sigma_{\#,\$}^{*}\#e_1\#e_2\#\cdots\#e_k\#\Sigma_{\#,\$}^{*}\#.$$

□

THEOREM 3.1(c). INCLUSION *is in co*NP *for*

$$RE(all - \{(+a)^{*},(+w)^{*},(+a)^{+},(+w)^{+}\}).$$

Proof. Let r_1, r_2 be expressions in $\mathrm{RE}(\mathrm{all} - \{(+a)^{*},(+w)^{*},(+a)^{+},(+w)^{+}\})$. It is clear that there is a number N_{r_1,r_2}, at most exponential in $|r_1| + |r_2|$ such that there is a DFA for $L(r_1) - L(r_2)$ of size at most N_{r_1,r_2}. In particular, if $L(r_1) \not\subseteq L(r_2)$, there is a counterexample string s of length at most N_{r_1,r_2}.

We show next that, because of the restricted form of r_1, it is possible to encode strings $s \in L(r_1)$ of length up to N_{r_1,r_2} as compressed strings of size polynomial in $|r_1|$. 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 $w_1,\dots,w_k \in \Sigma^{+}$. Hence, s can be written as $s_1 \cdots s_n$, where each s_i matches e_i. If e_i is of the form $(w_1^{*}+\cdots+w_k^{*})$ or $(w_1^{+}+\cdots+w_k^{+})$, then $s_i = w_j^{\ell}$, for some j,ℓ where $\ell \leq N_{r_1,r_2}$. So, the binary representation of ℓ has polynomial length in $|r_1|$. We can therefore represent s_i

by the pair (w_j, ℓ). In all other cases, $|s_i| \leq |e_i|$ and we represent s_i simply by $(s_i, 1)$. To check $L(r_1) \not\subseteq L(r_2)$, it thus suffices to guess a compressed string of polynomial length and to check that it matches r_1 but not r_2. We conclude that testing inclusion is in coNP. ∎

Notice that, in the proof of Theorem 3.1(c), we actually did not make use of the restricted structure of the expression r_2. We can therefore state the following corollary:

COROLLARY 3.5. *Let r_1 be an RE(all–$\{(+a)^*, (+w)^*, (+a)^+, (+w)^+\}$) expression and let r_2 be an arbitrary regular expression. Then, deciding whether $L(r_1) \subseteq L(r_2)$ is in coNP.*

Theorem 3.1(d) holds as even the containment problem for arbitrary regular expressions is in PSPACE. This concludes the proof of Theorem 3.1.

3.2. 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 RE$(a, a?)$ and 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 RE$(a, a?)$ and RE(a, a^*, a^+) expressions.

We now define the required normal form for RE$(a, a?)$ and 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 $s[0, *]$; and,
- 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 $aa?aa?b^*bb?b^*$ 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 $s[i, j]$ a *factor of* r'.

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

$$r_1(a, b) = a[i, *]b[0, *]a[0, *]b[1, *]a[l, *]$$

and

$$r_2(a, b) = a[i, *]b[1, *]a[0, *]b[0, *]a[l, *]$$

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 3.6. EQUIVALENCE *is in* PTIME *for*

(1) $RE(a, a?)$*, and*

(2) $RE(a, a^*, a^+)$.

Proof. We need to 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 $\max(r)$ for the maximum number in r different from $*$, that is, for $\max\{\{\text{upp}(r_i) \mid \text{upp}(r_i) \neq *\} \cup \{\text{low}(r_i) \mid 1 \leq i \leq n\}\}$. (We need the $\text{low}(r_i)$ the case that all upper bounds are $*$.) 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 $k := 1 + \max(\max(r'), \max(s'))$, 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 $v^{\max} = v_1^{\max} \cdots v_n^{\max}$, where, for every $i = 1, \ldots, n$,

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

Obviously, $v^{\max}$ is an element of $L(r)$. Hence, by our assumption that r is equivalent to s, we also have that $v^{\max} \in L(s)$. As $v^{\max}$ contains n blocks, s' must have at least n factors, so $m \geq n$. Correspondingly, we define the string $w^{\max} = w_1^{\max} \cdots w_m^{\max}$, where, for every $j = 1, \ldots, m$,

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

As $w^{\max}$ 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 $v_i^{\max}$ 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 $v^{\geq 1} = v_1^{\geq 1} \cdots v_n^{\geq 1}$, in which we have that each

$v_i^{\geq 1} = \text{base}(r_i)^{\max(\text{low}(r_i),1)}$ and its counterpart $w^{\geq 1} = w_1^{\geq 1} \cdots w_n^{\geq 1}$, in which each $w_i^{\geq 1} = \text{base}(s_i)^{\max(\text{low}(s_i),1)}$, it is immediate that, for each $i = 1, \ldots, n$, $\text{low}(s_i) = \text{low}(r_i)$ whenever $\text{low}(s_i) \geq 2$ or $\text{low}(r_i) \geq 2$. In addition, if we consider the string $v^{\min} = v_1^{\min} \cdots v_n^{\min}$, for which each $v_i^{\min} = \text{base}(r_i)^{\text{low}(r_i)}$ and its counterpart $w^{\min} = w_1^{\min} \cdots w_n^{\min}$, in which each $w_i^{\min} = \text{base}(s_i)^{\text{low}(s_i)}$, it is immediate that, between every pair of lower bounds which are at least two, the sequences of base symbols with lower bound one are the same in r' and s'. Similarly, we immediately have that

- before the leftmost lower bound which is at least two, and
- after the rightmost lower bound which is at least two,

the sequences of lower bounds which are zero or one are 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. Without loss of generality, we can assume that there is no index j with $\text{low}(s_j) < \text{low}(r_j)$ and $j < i_{\min}$. We consider two cases.

Case 1: If $\text{low}(r_{i_{\min}}) > 0$, we define the string v' by replacing $v_{i_{\min}}^{\max}$ in $v^{\max}$ by the sequence $\text{base}(r_{i_{\min}})^{\text{low}(r_{i_{\min}})}$. As $\text{low}(s_{i_{\min}}) > \text{low}(r_{i_{\min}})$ this string can not 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.

Case 2: In the second case, assume that $0 = \text{low}(r_{i_{\min}}) < \text{low}(s_{i_{\min}})$. As shown above, if $\text{low}(s_{i_{\min}}) \geq 2$ then $\text{low}(r_{i_{\min}}) \geq 2$. Hence, the only possibility is that $0 = \text{low}(r_{i_{\min}}) < \text{low}(s_{i_{\min}}) = 1$.

- If $i_{\min} = 1$, then the string $v_2^{\max} \cdots v_n^{\max}$ 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 $v_1^{\max} \cdots v_{n-1}^{\max}$ 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 $v_1^{\max} \cdots v_{i_{\min}-1}^{\max} v_{i_{\min}+1}^{\max} \cdots v_n^{\max}$ 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 $v_j^{\max}$, for some $j > i_{\min} + 1$ or $j < i_{\min} - 1$. But then all the blocks before $v_j^{\max}$ or after $v_j^{\max}$ (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 $v_{i_{\min}-1}^{\max} v_{i_{\min}+1}^{\max}$ as the length of this string is $\text{upp}(r_{i_{\min}-1}) + \text{upp}(r_{i_{\min}+1}) = \text{upp}(s_{i_{\min}-1}) + \text{upp}(s_{i_{\min}+1})$, which is strictly more than $\max(\text{upp}(s_{i_{\min}-1}), \text{upp}(s_{i_{\min}+1}))$. This would mean again that $s_{i_{\min}}$ can only match $v_j^{\max}$, for some $j > i_{\min} + 1$ or $j < i_{\min} - 1$. But this again implies that all the blocks before $v_j^{\max}$ or after $v_j^{\max}$ (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 again have that s can not match x, a contradiction.

(b) Now let r and s be from $\text{RE}(a, a^*)$. Let $j_{\min} > i_{\min}$ be minimal such that $\text{low}(r_{j_{\min}}) > 0$. 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,

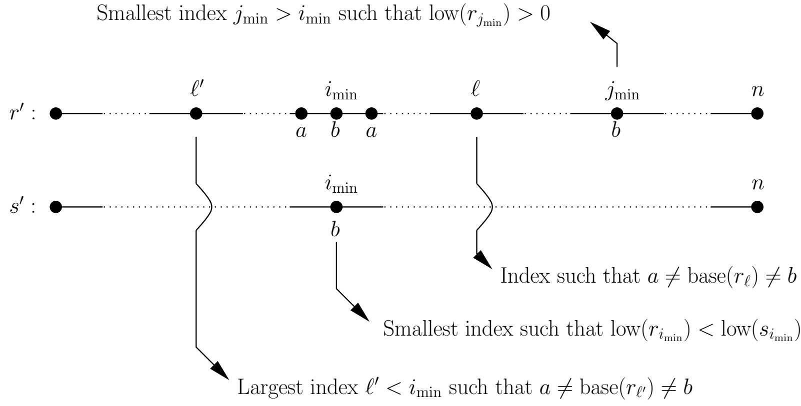


FIG. 3.2. *Graphical representation of several indices defined in the proof of Theorem 3.6*

- $\text{base}(r_{j_{\min}}) = \text{base}(r_{i_{\min}}) = \text{base}(s_{i_{\min}})$, and
- $\text{low}(r_{j_{\min}}) = \text{low}(s_{i_{\min}}) = 1$.

Let b be the symbol $\text{base}(r_{j_{\min}}) = \text{base}(r_{i_{\min}})$ and let a be $\text{base}(r_{i_{\min}-1}) = \text{base}(r_{i_{\min}+1})$. Notice that $a \neq b$ due to the definition of sequence normal form.
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.
Towards a contradiction, assume that there is some ℓ, $i_{\min} < \ell < j_{\min}$, 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 ℓ', we set $\ell' = 0$. We present the ordering of the defined indices $i_{\min}, j_{\min}, \ell'$, and ℓ in Figure 3.2. Recall that, for each $d = 1, \ldots, n$, $v_d^{\geq 1}$ was defined by

$$v_d^{\geq 1} = \begin{cases} \text{base}(r_d)^1 & \text{if } \text{low}(r_d) = 0, \\ \text{base}(r_d)^{\text{low}(r_d)} & \text{otherwise.} \end{cases}$$

Let $x' = v_1^{\geq 1} \cdots v_{\ell'}^{\geq 1} v_{\ell'+1}^{\min} \cdots v_{\ell-1}^{\min} v_\ell^{\geq 1} \cdots v_n^{\geq 1}$. As x' matches r, x' also matches s. Note that, in this match, $v_{\ell'}^{\geq 1}$ has to be matched by $s_{\ell'}$, because the string $v_1^{\geq 1} \cdots v_{\ell'}^{\geq 1}$ contains ℓ' blocks. Analogously, $v_\ell^{\geq 1}$ has to be matched by s_ℓ because $v_\ell^{\geq 1} \cdots v_n^{\geq 1}$ contains $n - \ell + 1$ blocks.
As $i_{\min}$ is minimal such that $\text{low}(r_{i_{\min}}) < \text{low}(s_{i_{\min}})$, as $\text{base}(r_{i_{\min}}) = b$, and as all factors f in $r_{i_{\min}} \cdots r_\ell$ have $\text{low}(f) = 0$, 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 $v_{\ell'}^{\geq 1}$ and $v_\ell^{\geq 1}$ is smaller than the sum of the numbers $\text{low}(f)$ over the factors f in $s_{\ell'+1} \cdots s_{\ell-1}$ with $\text{base}(f) = b$. This contradicts the fact that x' matches s. We can conclude that there is no such ℓ, 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 $\text{base}(r_{j_{\min}+1})$ exists and is different from a and b, say $\text{base}(r_{j_{\min}+1}) = c$. If $\text{low}(s_{j_{\min}}) = 0$ then the string $v_1^{\geq 1} \cdots v_{j_{\min}-1}^{\geq 1} v_{j_{\min}+1}^{\geq 1} \cdots v_n^{\geq 1}$ (consisting of $n-1$ blocks, as c is different from a or b) matches s, because $v_1^{\geq 1} \cdots v_n^{\geq 1}$ matches s. However, $v_1^{\geq 1} \cdots v_{j_{\min}-1}^{\geq 1} v_{j_{\min}+1}^{\geq 1} \cdots v_n^{\geq 1}$ does not match r because $\text{low}(r_{j_{\min}}) > 0$. This contradicts that r and s are

equivalent. Let ℓ' be defined as before. If $\text{low}(s_{j_{\min}}) \geq 1$ then the string $v_1^{\geq 1} \cdots v_{\ell'}^{\geq 1}$ $v_{\ell'+1}^{\min} \cdots v_{j_{\min}}^{\min} v_{j_{\min}+1}^{\geq 1} \cdots v_n^{\geq 1}$ matches r but not s (as it has too few bs between ℓ' and $j_{\min}$). Analogously, we can obtain that $j_{\min} < n$. Hence, $r_{j_{\min}+1}$ exists and $\text{base}(r_{j_{\min}+1}) = a$, because $\text{base}(r_{j_{\min}}) = b$.
We still need to deal with the case where $\text{base}(r_{j_{\min}+1}) = a$. Hence, between position $i_{\min} - 1$ and $j_{\min} + 1$, r' consists of a sequence of factors f which alternate between $\text{base}(f) = a$ and $\text{base}(f) = b$, and ends with the factor $r_{j_{\min}+1}$ for which $\text{base}(r_{j_{\min}+1}) = a$. Furthermore,

- all the factors $f = r_{i_{\min}}, \ldots, r_{j_{\min}-1}$ have $\text{low}(f) = 0$,
- $\text{low}(r_{i_{\min}-1}) = \text{low}(s_{i_{\min}-1})$, and
- $\text{low}(r_{j_{\min}}) = \text{low}(s_{i_{\min}}) = 1$.

It follows immediately that r' is not the sequence normal form of r as it contains a subsequence of the form $a[i, *]b[0, *]a[0, *]b[1, *]a[l, *]$.

□

3.3. Intersection. For arbitrary regular expressions, INTERSECTION is PSPACE-complete. We show that the problem is NP-hard for the same seemingly innocent fragments $\text{RE}(a, a^*)$, $\text{RE}(a, a?)$, $\text{RE}(a, (+a^+))$ and $\text{RE}(a^+, (+a))$ already studied in Section 3.1. By $\text{RE}(\text{all} - \{(+w)^*, (+w)^+\})$, we denote the fragment of RE(all) where we do not allow factors of the form $(w_1 + \cdots + w_n)^*$ or $(w_1 + \cdots + w_n)^+$ for which

- $n \geq 2$ and
- there is a $1 \leq j \leq n$ such that the length of w_j is at least two.

In other words, we still allow factors of the form $(a_1 + \cdots + a_n)^*$ and $(a_1 + \cdots + a_n)^+$ with single-label disjuncts but we exclude factors in which some disjunct has at least two symbols. For the fragment $\text{RE}(\text{all} - \{(+w)^*, (+w)^+\})$, we obtain a matching NP-upper bound. INTERSECTION is already PSPACE-hard for $\text{RE}(a, (+w)^*)$ and $\text{RE}(a, (+w)^+)$ expressions. This follows from a result of Bala, who showed that it is PSPACE-hard to decide whether the intersection of an arbitrary number of $\text{RE}((+w)^*)$ expressions contains a *non-empty* string [4]. Compared to the fragments for which we settled the complexity of INCLUSION in Theorem 3.1 in Section 3.1, we only need to leave the precise complexity of $\text{RE}(a, w^+)$ open, i.e., we only prove an NP upper bound. These results are summarized in the following theorem:

THEOREM 3.7.
(a) INTERSECTION *is* NP-*hard for*
(1) $\text{RE}(a, a^*)$*;*
(2) $\text{RE}(a, a?)$*;*
(3) $\text{RE}(a, (+a)^+)$*;*
(4) $\text{RE}(a, (+a^+))$*; and*
(5) $\text{RE}(a^+, (+a))$*;*
(b) INTERSECTION *is in* NP *for* $\text{RE}(\text{all} - \{(+w)^*, (+w)^+\})$*;*
(c) INTERSECTION *is* PSPACE-*hard for*
(1) $\text{RE}(a, (+w)^*)$*; and*
(2) $\text{RE}(a, (+w)^+)$*; and,*
(d) INTERSECTION *is in* PSPACE *for* $\text{RE}(\text{all})$.

As in Section 3.1, we split the proof of Theorem 3.7 into several parts to improve readability.

We start by proving Theorem 3.7(a). This proof is along the same lines as the proof of Theorem 3.1(a). However, we would like to point out that the INTERSECTION problem is not simply dual to the INCLUSION problem. This is witnessed by, for

instance, the fragments $\mathrm{RE}(a, (+a)^*)$ and $\mathrm{RE}(a, (+a)^+)$ for which INTERSECTION is NP-complete, while INCLUSION is PSPACE-complete.

In all five cases, it is a LOGSPACE reduction from SATISFIABILITY of propositional 3CNF formulas. The SATISFIABILITY problem asks, given a propositional formula Φ in 3CNF with variables $\{x_1, \dots, x_n\}$, whether there exists a truth assignment for $\{x_1, \dots, x_n\}$ for which Φ is true. The SATISFIABILITY problem for 3CNF formulas is known to be NP-complete [18]. We note that, for the cases (1)–(3), INTERSECTION is already NP hard when the expressions use a fixed-size alphabet.

Let $\Phi = C_1 \wedge \cdots \wedge C_k$ be a 3CNF formula using variables $\{x_1, \dots, x_n\}$. Let, for each i, $C_i = L_{i,1} \vee L_{i,2} \vee L_{i,3}$ be the ith clause. Our goal is to construct regular expressions $R_1, \dots, R_k$ and S_1, S_2 such that

$$L(R_1) \cap \cdots \cap L(R_k) \cap L(S_1) \cap L(S_2) \neq \emptyset \text{ if and only if } \Phi \text{ is satisfiable.} \qquad (\dagger)$$

Analogously as in the proof of Theorem 3.1(a), we encode truth assignments for Φ by strings. We construct S_1 and S_2 such that $L(S_1 \cap S_2)$ contains all syntactically correct encodings and a string w matches R_i if and only if w represents an assignment which makes C_i true.

To define S_1, S_2 and $R_1, \dots, R_k$ we use auxiliary regular expressions W_1, W_2, and N, a string u, and expressions $F(L_{i,j})$ associated with literals $L_{i,j}$:

- The strings of $L(W_1) \cap L(W_2)$ will be interpreted as truth assignments. 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.
- For each literal $L_{i,j}$ and every $w \in L(W_1) \cap L(W_2)$ it will hold that $w \models F(L_{i,j})$ if and only if $A_w \models L_{i,j}$.
- The string u matches every expression $F(L_{i,j})$.
- Finally, N can match the concatenation of up to 3 occurrences of the string u.

Then we define

$$\begin{aligned} S_1 &= u^3 W_1 u^3, \\ S_2 &= u^3 W_2 u^3, \\ R_i &= N F(L_{i,1}) F(L_{i,2}) F(L_{i,3}) N, \end{aligned}$$

for $i = 1, \dots, k$. Intuitively, $L(R_1) \cap \cdots \cap L(R_k) \cap L(S_1) \cap L(S_2) \neq \emptyset$ now holds if and only if every truth assignment (matched by the middle parts of S_1 and S_2) makes at least one literal true in each conjunct (i.e., matches at least one $F(L_{i,j})$ for every $i = 1, \dots, k$).

For each of the cases (1)–(5), we construct the auxiliary expressions with the help of five basic regular expressions r^{true}, r^{false}, $r^{\text{true/false}}$, r^{all}, and α, which must adhere to the inclusion structure graphically represented in Figure 3.3 and formally stated by the following properties (INT1)–(INT4).

$$\alpha \in L(r^{\text{false}}) \cap L(r^{\text{true}}) \cap L(r^{\text{all}}) \qquad \text{(INT1)}$$

$$z^{\text{false}} \in L(r^{\text{false}}) \cap L(r^{\text{true/false}}) \cap L(r^{\text{all}}) \qquad \text{(INT2)}$$

$$z^{\text{true}} \in L(r^{\text{true}}) \cap L(r^{\text{true/false}}) \cap L(r^{\text{all}}) \qquad \text{(INT3)}$$

$$L(r^{\text{false}}) \cap L(r^{\text{true}}) \cap L(r^{\text{true/false}}) = \emptyset \qquad \text{(INT4)}$$

Intuitively, the two dots in Figure 3.3 are strings z^{true} and z^{false}, which represent the truth values *true* and *false*, respectively. The expressions r^{true} and r^{false} are

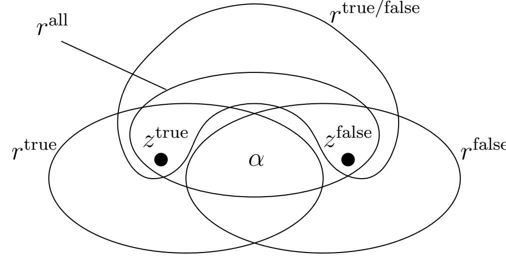


Fig. 3.3. *Inclusion structure of regular expressions used in* NP*-hardness of* INTERSECTION.

	(1) RE(a, a^*)	(2) RE$(a, a?)$	(3) RE$(a, (+a)^+)$
α	a	a	a
r^{true}	aa^*b^*	$ab?$	$(a+b)^+$
r^{false}	b^*aa^*	$b?a$	$(a+c)^+$
$r^{\text{true/false}}$	$aa^*bb^* + bb^*aa^*$	$ab + ba$	$(b+c)^+$
$s^{\text{true/false}}$	$b^*aa^*b^*$	$b?ab?$	
$(s')^{\text{true/false}}$	$a^*bb^*a^*$	$a?ba?$	
r^{all}	$a^*b^*a^*$	$a?b?a?$	$(a+b+c)^+$
z^{true}	ab	ab	b
z^{false}	ba	ba	c
W_1	$\#s^{\text{true/false}}\$\cdots$ $\cdots\$s^{\text{true/false}}\#$	$\#s^{\text{true/false}}\$\cdots$ $\cdots\$s^{\text{true/false}}\#$	$\#r^{\text{true/false}}\$\cdots$ $\cdots\$r^{\text{true/false}}\#$
W_2	$\#(s')^{\text{true/false}}\$\cdots$ $\cdots\$(s')^{\text{true/false}}\#$	$\#(s')^{\text{true/false}}\$\cdots$ $\cdots\$(s')^{\text{true/false}}\#$	$\#r^{\text{true/false}}\$\cdots$ $\cdots\$r^{\text{true/false}}\#$
N	$(\#^*a^*\$^*\cdots\$^*a^*\#^*)^3$	$(\#?a?\$?\cdots\$?a?\#?)^3$	$(\#+\$+a)^+$
u	$\#\alpha\$\cdots\$\alpha\#$	$\#\alpha\$\cdots\$\alpha\#$	$\#\alpha\$\cdots\$\alpha\#$

Table 3.3

Definition of basic and auxiliary expressions for cases (1)–(3). In the definition of W_1, W_2, N and u the repetition indicated by $\cdots$ is always n-fold and does not include $\#$.

used to match z^{true} and z^{false} in the $F(L_{i,j})$, respectively. The expression $r^{\text{true/false}}$ is defined through the intersection of W_1 and W_2 to generate both *true* and *false*. Finally, α is used in u and r^{all} for the definition of F.

The definition of the basics expressions and of W, N and u for cases (1)–(3) is given in Table 3.3. We will consider cases (4) and (5) later.

It is straighforward to verify that the conditions (INT1)–(INT4) are fulfilled for each of the fragments. Note that the expressions use the fixed alphabet $\{a, b, c, \$, \#\}$.

The association between strings and truth assignments is now straightforward.

- For $w = \#w_1\$\cdots\$w_n\# \in L(W_1) \cap L(W_2)$ let A_w be defined as follows:

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

- For a truth assignment A, let $w_A = \#w_1\$\cdots\$w_n\#$, where, for each $j =$

$1, \ldots, n$,

$$w_j = \begin{cases} z^{\text{true}} & \text{if } A(x_j) = \text{true} \\ z^{\text{false}} & \text{otherwise.} \end{cases}$$

- Finally, for each literal $L_{i,j}$, we define $F(L_{i,j}) = \#e_1\$ \cdots \$e_n\#$, where for each $\ell = 1, \ldots, n$,

$$e_\ell := \begin{cases} r^{\text{true}}, & \text{if } L_{i,j} = x_\ell, \\ r^{\text{false}}, & \text{if } L_{i,j} = \neg x_\ell, \text{ and} \\ r^{\text{all}}, & \text{otherwise.} \end{cases}$$

Notice that only one e_ℓ among $\{e_1, \ldots, e_n\}$ is different from r^{all}.

Claim 3.8. *If $A \models L_{i,j}$ then $w_A \in L(F(L_{i,j}))$. If $w \in L(F(L_{i,j}))$ then $A_w \models L_{i,j}$.*

Proof of Claim 3.8. Let A be a truth assignment such that $A \models L_{i,j}$ and let $w_A = \#w_1\$ \cdots \$w_n\#$ be defined as above. Let $F(L_{i,j}) = \#e_1\$ \cdots \$e_n\#$ be defined as above. We need to show that $w_A \in L(F(L_{i,j}))$. To this end, let $\ell \leq n$. There are three cases to consider:

1. If $L_{i,j} = x_\ell$ then $A(x_\ell) = \text{true}$. Hence, as $w_\ell = z^{\text{true}}$, as $e_\ell = r^{\text{true}}$, and by condition (INT3), we have $w_\ell \in L(e_\ell)$.
2. If $L_{i,j} = \neg x_\ell$ then $A(x_\ell) = \text{false}$. Hence, as $w_\ell = z^{\text{false}}$, as $e_\ell = r^{\text{false}}$, and by condition (INT2), we have $w_\ell \in L(e_\ell)$.
3. Otherwise, we have that w_ℓ is either z^{true} or z^{false}. As $e_\ell = r^{\text{all}}$ we have that $w_\ell \in L(e_\ell)$ by (INT3) in the former case and by (INT2) in the latter case.

Therefore, for each $\ell = 1, \ldots, n$, $w_\ell \in L(e_\ell)$ and thus $w_A \in L(F(L_{i,j}))$.

We show the other statement by contraposition. Thereto, let $w \in L(W_1) \cap L(W_2)$ be a string such that A_w is a truth assignment that does not make literal $L_{i,j}$ true. We show that $w \notin L(F(L_{i,j}))$. Let $F(L_{i,j}) = \#e_1\$ \cdots \$e_n\#$ as above. There are two cases:

1. Suppose that $L_{i,j} = x_\ell$ for some $1 \leq \ell \leq n$. Then $A_w(x_\ell) = \text{false}$ and therefore, $w_\ell \notin L(r^{\text{true}})$. By definition of $F(L_{i,j})$, we have $e_\ell = r^{\text{true}}$ and therefore $w_\ell \notin L(e_\ell)$. Hence, $w \notin L(F(L_{i,j}))$.
2. Suppose that $L_{i,j} = \neg x_\ell$ for some $1 \leq \ell \leq n$. Then $A_w(x_\ell) = \text{true}$ and therefore, $w_\ell \in r^{\text{true}}$. By definition of $F(L_{i,j})$, we have $e_\ell = r^{\text{false}}$. As $w \in L(W_1) \cap L(W_2)$ we have that $w_\ell \in r^{\text{true/false}}$. Because of (INT4) we therefore have that $w_\ell \notin r^{\text{false}}$ and therefore $w_\ell \notin L(e_\ell)$. Hence, $w \notin L(F(L_{i,j}))$. ∎

In the proof of (†) we will make use of the following properties.

Claim 3.9.

(a) $u^i \in L(N)$ for every $i = 1, \ldots, 3$.
(b) $u \in L(F(L_{i,j}))$ for every $i = 1, \ldots, k$ and $j = 1, 2, 3$.
(c) If $u^3 w u^3 \in L(S_1) \cap L(S_2) \cap L(R_1) \cap \cdots \cap L(R_k)$, then w matches some $F(L_{i,j})$ in every R_i.

Proof of Claim 3.9. (a) can be easily checked and (b) follows immediately from condition (INT1) and the definition of F. In the following, we abbreviate $F(L_{i,j})$ by $F_{i,j}$. To show (c), suppose that $u^3 w u^3 \in L(S_1) \cap L(S_2) \cap L(R_1) \cap \cdots \cap L(R_k)$. We need to show that w matches some $F(L_{i,j})$ in every R_i.

Observe that the strings, u, w, and every string in $L(F_{i,j})$ is of the form $\#y\#$, where y is a non-empty string over alphabet $\{a, b, c, \$\}$. Hence, as for every $i =$

$1, \ldots, k$, $u^3wu^3 \in L(R_i)$, and as none of the strings y 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 u^3wu^3 and R_i. Let $y_1, \ldots, y_7$ be such that $\#y_1\#\#y_2\#\#y_3\#\#y_4\#\#y_5\#\#y_6\#\#y_7\# = u^3wu^3$. Let ℓ be maximal such 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 $\ell \leq 3$ by definition of N. We now prove that m matches w onto some $F_{i,j}$. 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 $\ell \geq 1$ by definition of N. Again, towards a contradiction, assume that m matches w onto the right occurrence of N in R_i. Observe that any string that matches $NF_{i,1}F_{i,2}F_{i,3}$ 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 $NF_{i,1}F_{i,2}F_{i,3}$, 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 $j = 1, 2, 3$. ☐

Now we can complete the proof of (†) for fragments (1)–(3).

Proof of (†). ($\Rightarrow$) Assume that $S_1 \cap S_2 \cap \bigcap_{i=1}^k R_i \neq \emptyset$. Hence, there exists a string $v = u^3wu^3$ in $S_1 \cap S_2 \cap \bigcap_{i=1}^k R_i$. By Claim 3.9, w matches some $F(L_{i,j})$ in every R_i. By Claim 3.8, $A_w \models L_{i,j}$ for every $i = 1, \ldots, k$. Hence, Φ is true under truth assignment A_w, so Φ is satisfiable.

($\Leftarrow$) Suppose now that Φ is true under some truth assignment A. Hence, for every i, some $L_{i,j}$ becomes true under A and therefore $w_A \in L(F(L_{i,j}))$ by Claim 3.8. As u, u^2, and u^3 are in $L(N)$ and as u is in each $L(F(L_{i,j}))$ by Claim 3.9, we get that the string $u^3w_Au^3$ is in $L(S_1) \cap L(S_2) \cap L(R_1) \cap \cdots \cap L(R_k)$. ☐

This completes the proof of Theorem 3.7(a) for the fragments (1)–(3).

We still need to deal with the fragments (4) and (5). As in the proof of Theorem 3.1(a), we will no longer use an alphabet with fixed size. Instead, we use symbols b_j and c_j for $j = 1, \ldots, n$. Instead of the basic regular expressions r^{true}, r^{false}, $r^{\text{true/false}}$, and r^{all}, we will now have expressions r_j^{true}, r_j^{false}, $r_j^{\text{true/false}}$, and r_j^{all} for every $j = 1, \ldots, n$. We will require that these expressions have the same properties (INT1)–(INT4), but only between the expressions r_j^{true}, r_j^{false}, $r_j^{\text{true/false}}$, and r_j^{all} with the same index j, and α. The definitions of the basic and auxiliary expressions are given in Table 3.4.

The association between strings and truth assignments is now defined as follows.

- For $w = w_1 \cdots w_n \in L(W_1) \cap L(W_2)$ let A_w be defined as follows

$$A_w(x_j) := \begin{cases} \text{true}, & \text{if } w_j \in L(r_j^{\text{true}}), \\ \text{false}, & \text{otherwise.} \end{cases}$$

- For a truth assignment A, let $w_A = w_A = w_1 \cdots w_n$ where, for each $j = 1, \ldots, n$,

$$w_j := \begin{cases} z_j^{\text{true}} & \text{if } A(x_j) = \text{true} \\ z_j^{\text{false}} & \text{otherwise.} \end{cases}$$

	(4) RE$(a, (+a^+))$	(5) RE$(a^+, (+a))$
α	a	a
r_j^{true}	$(a^+ + b_j^+)$	$(a + b_j)$
r_j^{false}	$(a^+ + c_j^+)$	$(a + c_j)$
$r_j^{\text{true/false}}$	$(b_j^+ + c_j^+)$	$(b_j + c_j)$
r_j^{all}	$(a^+ + b_j^+ + c_j^+)$	$(a + b_j + c_j)$
z_j^{true}	b_j	b_j
z_j^{false}	c_j	c_j
$W_1 (= W_2)$	$r_1^{\text{true/false}} \cdots r_n^{\text{true/false}}$	$r_1^{\text{true/false}} \cdots r_n^{\text{true/false}}$
N	a^+	a^+
u	α^n	α^n

Table 3.4
Definition of basic and auxiliary expressions for cases (4)–(5).

- Finally, for each literal $L_{i,j}$, we define $F(L_{i,j}) = e_1 \cdots e_n$,, where, for each $\ell = 1, \ldots, n$,

$$e_\ell := \begin{cases} r_\ell^{\text{false}}, & \text{if } L_{i,j} = \neg x_\ell, \\ r_\ell^{\text{true}}, & \text{if } L_{i,j} = x_\ell,\text{, and} \\ r_\ell^{\text{all}}, & \text{otherwise.} \end{cases}$$

Notice that only one e_ℓ among $\{e_1, \ldots, e_n\}$ is different from r_ℓ^{all}.

In order to prove (†) for fragments (4) and (5), we have to show that Claims 3.8 and 3.9 hold. For Claim 3.8 this can be done similarly as before, and for Claim 3.9(a) and (b) it is again easy to see. We complete the proof of Theorem 3.7(a) by showing Claim 3.9(c).

To this end, suppose that $u^3wu^3 \in L(S_1) \cap L(S_2) \cap L(R_1) \cap \cdots \cap L(R_k)$. 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 u^3wu^3 and R_i, for every $i = 1, \ldots, k$. For every $\ell = 1, \ldots, n$, let Σ_ℓ denote the set $\{b_\ell, c_\ell\}$. Observe that the string w is of the form $y_1 \cdots y_n$, where, for every $\ell = 1, \ldots, n$, y_ℓ is a string in Σ_ℓ^+. Moreover, no strings in $L(N)$ contain symbols from Σ_ℓ for any $\ell = 1, \ldots, n$. 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 $y'_1 \ldots y'_n$, where each y'_ℓ is a string in $(\Sigma_\ell \cup \{a\})^+$. As m_i can only match symbols in Σ_ℓ onto subexpressions with symbols in Σ_ℓ, m_i matches w onto some $F_{i,j}$.

This completes the proof of Theorem 3.7(a). ∎

The following part of Theorem 3.7 makes the difference between INCLUSION and INTERSECTION apparent. Indeed, INCLUSION for RE$(\text{all} - \{(+w)^*, (+w)^+\})$ 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 RE$^{\leq k}$ is in PTIME. Later in this section, we will show that INTERSECTION is PSPACE-hard for RE$^{\leq 3}$ expressions (Theorem 3.10).

THEOREM 3.7(b). INTERSECTION *is in* NP *for RE*$(\mathit{all} - \{(+w)^*, (+w)^+\})$.

Proof. Let $r_1, \ldots, r_k$ be RE(all $- \{(+w)^*, (+w)^+\}$) 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 $xw^\ell y$ for some ℓ. In the latter case, this substring can be written as $(x,1)(w,\ell)(y,1)$, for a suitable ℓ, 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 2.4.

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 ε. Let, for each i, $r_i = e_{i,1} \cdots e_{i,n'}$ where every $e_{i,j}$ is a factor.

Let $u = a_1 \cdots a_{\min}$ 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 a unique pair (ℓ, ℓ') such that $u[\ell, \ell'] \models_{m_i} e_{i,j}$ and such that, for all other pairs (ℓ_0, ℓ_0') with $u[\ell, \ell'] \models_{m_i} e_{i,j}$ we have $\ell \leq \ell_0 \leq \ell_0' + 1 \leq \ell'$. That is, there is a unique pair (ℓ, ℓ') that subsumes all others. In the remainder of this proof, we only consider such pairs (ℓ, ℓ'). We call a pair (p, p') of positions *homogeneous*, if, for each i, there are ℓ_i, ℓ_i' and j such that $\ell_i \leq p$, $p' \leq \ell_i'$ and $u[\ell_i, \ell_i'] \models_{m_i} e_{i,j}$. Intuitively, a pair is homogeneous if, for each i, all its symbols are subsumed by the same subexpression $e_{i,j}$. We call a pair (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 $\ell_0 = 0, \ell_{\text{maxpos}} = \min$ (i.e., the last position in u), 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 $p = 1, \ldots, \text{maxpos}$, u_p denote the substring $u[\ell_{p-1} + 1, \ell_p]$ of u. We consider each substring u_p separately and distinguish the following cases:

- If u_p is a substring of $u[\ell, \ell']$ for which there is at least one m_i with $u[\ell, \ell'] \models_{m_i} x$, where x is of the form a, $a?$, w, $w?$, $(+a)$, $(+a)?$, $(+w)$, or $(+w)?$ (in abbreviated notation), then $|u_p| \leq |e| \leq n$. We define $v_p = u_p$.
- If there exist, for each i, ℓ and ℓ' such that u_p is a substring of every $u[\ell, \ell']$ and $u[\ell, \ell'] \models_{m_i} x_i$ for every m_i, where each x_i is either of the form $(+a)^*$ or $(+a)^+$ (in abbreviated notation) then u_p must have 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 $u[\ell, \ell']$ for which there is at least one m_i with $u[\ell, \ell'] \models_{m_i} x$, where x is of the form a^*, w^*, a^+, w^+, $(+a^*)$, $(+w^*)$, $(+a^+)$ or $(+w^+)$. Hence, u_p can be written as $xw^i y$ 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 $|v_p| = \mathcal{O}(n)$.

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

Bala has shown that deciding whether the intersection of RE($(+w)^*$) expressions contains a non-empty string is PSPACE-complete [4]. In general, the intersection problem for such expressions is trivial, as the empty string is always in the intersection. However, adapting Bala's proof to our INTERSECTION problem is just a technicality. We next present a 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 RE($a, (+w)^*$) and RE($a, (+w)^+$) expressions. It should be noted that the crucial idea in this proof comes from the proof of Bala [4].

THEOREM 3.7(c). INTERSECTION *is* PSPACE-*hard for*

(1) RE($a, (+w)^*$) *and*

(2) RE($a, (+w)^+$).

Proof. We first show that INTERSECTION is PSPACE-hard for RE($a, (+w)^+$) and we consider the case of RE($a, (+w)^*$) later. In both cases, we use a reduction from CORRIDOR TILING, which is PSPACE-complete [15].

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 BT, $\mathrm{Hor}_{\mathrm{even}}$, $\mathrm{Hor}_{\mathrm{odd}}$, $\mathrm{Ver}_1, \ldots, \mathrm{Ver}_n$ such that

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

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 $j = 1, \ldots, n$, we define $\Sigma_j := \{t_{i,j} \mid t_i \in T\}$, which we will use to tile the j-th column. We define $\Sigma := \bigcup_{1 \leq j \leq n} \Sigma_j$. For a set of Σ-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

$$\rhd(\triangle^n \Sigma_1 \triangle^n \Sigma_2 \triangle^n \cdots \triangle^n \Sigma_n \triangle^n \#)^* \lhd, \qquad (\star)$$

where $\rhd, \lhd, \triangle$, and $\#$ are special symbols not occurring in Σ. Here, we use the symbols "$\rhd$" and "$\lhd$" 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 "$\triangle$" 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 $\overline{b} = (\mathrm{bot}_1, \ldots, \mathrm{bot}_n)$ and $\overline{t} = (\mathrm{top}_1, \ldots, \mathrm{top}_n)$. In the expressions below, for each $i = 1, \ldots, n$, $\mathrm{bot}_{i,i}$ and $\mathrm{top}_{i,i}$ denote the copies of bot_i, resp., top_i in Σ_i.

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

$$\mathrm{BT} := \rhd \triangle^n \mathrm{bot}_{1,1} \triangle^n \cdots \triangle^n \mathrm{bot}_{n,n} \triangle^n \; (\Sigma \cup \{\#, \triangle\})^+ \; \triangle^n \mathrm{top}_{1,1} \triangle^n \cdots \triangle^n \mathrm{top}_{n,n} \triangle^n \# \lhd$$

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

$$\mathrm{Hor}_{\mathrm{even}} := \Bigg(\sum_{\substack{2\leq \ell\leq n-2\\ \ell \text{ is even}\\ (t_i,t_j)\in H}} (\triangle^n t_{i,\ell} \triangle^n t_{j,\ell+1}) + (\triangleright \triangle^n \mathrm{bot}_{1,1}) + (\triangle^n \mathrm{top}_{n,n} \triangle^n \# \triangleleft) + \sum_{\substack{1\leq i\leq k\\ 1\leq j\leq k}} (\triangle^n t_{i,n} \triangle^n \# \triangle^n t_{j,1}) \Bigg)^+ .$$

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 ℓ and $\ell+1$, where ℓ is an odd number between 1 and $n-1$. Together with $\mathrm{Hor}_{\mathrm{ver}}$, this expression also ensures that the strings in the intersection are correct encodings of tilings. (That is, the strings are in the language defined by $(\star)$.)

$$\mathrm{Hor}_{\mathrm{odd}} := \Bigg((\triangleright \triangle^n \mathrm{bot}_{1,1} \triangle^n \mathrm{bot}_{2,2}) + (\triangle^n \mathrm{top}_{n-1,n-1} \triangle^n \mathrm{top}_{n,n} \triangle^n \# \triangleleft) + \sum_{\substack{1\leq \ell\leq n-1\\ \ell \text{ is odd}\\ (t_i,t_j)\in H}} (\triangle^n t_{i,\ell} \triangle^n t_{j,\ell+1}) + (\triangle^n \#) \Bigg)^+ .$$

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

$$\mathrm{Ver}_\ell := \Bigg(\sum_{(\mathrm{bot}_{\ell,\ell},t_i)\in V} (\triangleright \triangle^n \mathrm{bot}_{1,1} \triangle^n \cdots \triangle^n \mathrm{bot}_{\ell,\ell} \triangle^{n-i}) + \sum_{\substack{1\leq i\leq n\\ 1\leq j\leq k\\ m\neq \ell}} (\triangle^i t_{j,m} \triangle^{n-i}) + \sum_{1\leq i\leq n} (\triangle^i \# \triangle^{n-i}) + \sum_{1\leq i\leq n} (\triangle^i \# \triangleleft) + \sum_{\substack{1\leq i\leq n\\ (t_i,t_j)\in V}} (\triangle^i t_{i,\ell} \triangle^{n-j}) \Bigg)^+ .$$

The crucial idea is that, when we read a tile in the ℓ-th column, we use the string $\triangle^{n-i}$ to encode that we want the tile in the ℓ-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 $\mathrm{Hor}_{\mathrm{odd}}$ and $\mathrm{Hor}_{\mathrm{even}}$. This way, we know that *(i)* the strings encode tilings in which every row consists of n tiles, *(ii)* we use symbols from Σ_j to encode tiles from the j-th column, and that *(iii)* the string $\triangle^n$ occurs between every two tiles. The

first disjunction allows us to get started on the bottom row. It says that we want the ℓ-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 ℓ-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 $\triangleleft$ only occurs once in each correctly encoded tiling. Finally, the fifth disjunction does the crucial step: it ensures that, when we matched the ℓ-th tile t_k in the previous row with an expression ending with $t_{k,\ell}\triangle^{n-i}$, 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 $\triangle^i t_{i,\ell}\triangle^{n-j}$ 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 ℓ-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_{even}, Hor_{odd}, $\text{Ver}_1, \ldots, \text{Ver}_n$, if and only if w encodes a correct tiling for D.

The PSPACE-hardness proof for $\text{RE}(a, (+w)^*)$ is completely analogous, except that every "+" (which stands as abbreviation for "one or more") needs to be replaced by the Kleene star "$*$". ☐

Theorem 3.7(d) holds as INTERSECTION for arbitrary regular expressions is in PSPACE. This concludes the proof of Theorem 3.7.

THEOREM 3.10. INTERSECTION *is* PSPACE-*complete for*

(a) one-unambiguous regular expressions; and for
(b) $RE^{\leq 3}$.

Proof. The PSPACE upper bound is immediate as INTERSECTION is in PSPACE for regular expressions in general.

We proceed by showing the lower bound. We reduce CORRIDOR TILING, which is PSPACE-complete [15], to both intersection problems. We first show that INTERSECTION is 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. 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.

Let $D = (T, H, V, \bar{b}, \bar{t}, n)$ be a tiling system. Without loss of generality, we assume that every correct tiling has at least four rows, an even number of rows, and that the tiles T are partitioned into three disjoint sets $T_0 \uplus T' \uplus T''$. The idea is that symbols from T'' are only used on the uppermost row and the symbols from T' only on the row below the uppermost row. 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, $\text{Hor-odd}_{t,i}$, $\text{Hor-even}_{t,i}$, $\text{Ver-odd}_{t,j}$, $\text{Ver-even}_{t,j}$, and $\text{Ver-last}_{t,j}$ for every $t \in T$, $i \in \{1, \ldots, n-1\}$, and $j \in \{1, \ldots, n\}$.

These expressions are constructed such that, for every string w, we have that

$$w \text{ encodes a correct tiling for } D \text{ if and only if}$$
$$w \in L(BT) \cap \bigcap_{t,i} \big(L(\text{Hor-odd}_{t,i}) \cap L(\text{Hor-even}_{t,i})\big)$$
$$\cap \bigcap_{t,j} \big(L(\text{Ver-odd}_{t,j}) \cap L(\text{Ver-even}_{t,j}) \cap L(\text{Ver-last}_{t,j})\big) \qquad (\Diamond)$$

For a set S of tiles, we denote by S the disjunction of all symbols in S whenever this improves readability. We also write S^i to indicate that S is used to tile the i-th column in our encoding. Furthermore, for $1 \le i \le j \le n$, we write $S^{i,j}$ to abbreviate $S^i \cdots S^j$. 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

  $$BT := b_1 \cdots b_n \Big(T_0^{1,n}\ \big(T_0^{1,n} + (T')^{1,n}\big)\Big)^* t''_1 \cdots t''_n$$

  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 $t \in T$ and each $j \in \{1, \ldots, n-1\}$, the expression $\text{Hor-odd}_{t,j}$ ensures that the right neighbor of t is correct in odd rows, where t is a tile in the j-th column.

  $$\text{Hor-odd}_{t,j} := \Big(T^{1,j-1}\big(t(s_1 + \cdots + s_\ell) + ((T - \{t\})T)\big)T^{j+2,n}T^{1,n}\Big)^*,$$

  where $s_1, \ldots, s_\ell$ are all tiles s_i with $(t, s_i) \in H$. Similarly, we define

  $$\text{Hor-even}_{t,j} := \Big(T^{1,n}T^{1,j-1}\big(t(s_1 + \cdots + s_\ell) + ((T - \{t\})T)\big)T^{j+2,n}\Big)^*.$$

  All expressions $\text{Hor-odd}_{t,j}$ and $\text{Hor-even}_{t,j}$ are 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 tile t: $\text{Ver-odd}_{t,j}$ checks the vertical constraints on all odd rows but the last, $\text{Ver-even}_{t,j}$ checks the vertical constraints on the even rows, and $\text{Ver-last}_{t,j}$ 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 $(t, s_i) \in V$.

  $$\text{Ver-odd}_{t,j} := T_0^{1,j-1}$$
  $$\Big[\big[(tT_0^{j+1,n}T_0^{1,j-1}(s_1 + \cdots + s_\ell)) + ((T_0 - \{t\})T_0^{j+1,n}T_0^{1,j})\big]T_0^{j+1,n}(T_0^{1,j-1} + T'^{1,j-1})\Big]^*$$
  $$T'^{j,n}T''^{1,n}$$

  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 $\text{Ver-odd}_{t,j}$. 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 $\text{Ver-odd}_{t,j}$ is one-unambiguous.

$$\begin{aligned}\text{Ver-even}_{t,j} := {} & T_0^{1,n} T_0^{1,j-1} \\ & \Big[\Big(\big(tT_0^{j+1,n}\big[(T_0^{1,j-1}(s_1 + \cdots + s_\ell)) + (T'^{1,j-1}(s'_1 + \cdots + s'_\ell))\big]\big) \\ & \qquad\qquad + ((T_0 - \{t\})T_0^{j+1,n}(T_0^{1,j} + T'^{1,j}))\Big) \\ & \qquad\qquad\qquad (T_0^{j+1,n} + T'^{j+1,n})(T_0^{1,j-1} + T''^{1,j-1})\Big]^* \\ & \qquad\qquad\qquad\qquad T''^{j,n}\end{aligned}$$

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 $\text{Ver-odd}_{t,j}$. Depending whether the n-th tile starting from t is on the row tiled 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 $\text{Ver-even}_{t,j}$ is one-unambiguous. Finally, we define the expressions $\text{Ver-last}_{t,j}$ as follows:

$$\begin{aligned}\text{Ver-last}_{t,j} := {} & (T_0^{1,n} T_0^{1,n})^* T'^{1,j-1} \\ & \big(t' T'^{j+1,n} T''^{1,j-1}(s''_1 + \cdots + s''_\ell)\big) + \big((T' - \{t'\})T'^{j+1,n}T''^{1,j}\big) \\ & \qquad T''^{j+1,n}\end{aligned}$$

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_{odd}, T^j_{even}, T'^j_{odd}, and T''^j_{even} for every $j = 1, \ldots, n$. Here, T'^j_{odd} and T''^j_{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, we now argue that the above defined regular expressions can be rewritten such that every tile of T occurs at most three times. The rewriting of the above expressions is simply the following: we only use symbols from T^j_{odd} for the j-th column of odd rows, and analogously for T^j_{even},

T'^{j}_{odd}, and T''^{j}_{even} for every $j = 1, \ldots, n$. For example, the leftmost occurrence of $T_0^{1,n}$ in BT is rewritten to $T^{1,n}_{\text{even}}$. In the arguments below, we still refer to the original notation in the expressions for readability.

- Expression BT: Each tile occurs at most twice. The only place where tiles occur more than once is where the tiles of $b_1, \ldots, b_n$ are repeated in the right occurrence in $T_0^{1,n}$. (The left occurrence of $T_0^{1,n}$ would contain only even-row tiles, which are different from the odd-row tiles $b_1, \ldots, b_n$.)
- Expression Hor-odd$_{t,j}$: Each tile occurs at most twice. The only place where tiles occur more than once is where tiles of $s_1, \ldots, s_\ell$ are repeated in the right occurrence of T (without superindex). Even-row tiles only occur in $T^{1,n}$ and the remainder of the expression only contains odd-row tiles.
- Expression Hor-even$_{t,j}$: This is similar to Hor-odd$_{t,j}$.
- Expression Ver-odd$_{t,j}$: Each tile occurs at most twice. The only places where tiles occur more than once are *(i)* in the leftmost occurrence of $T_0^{1,j-1}$ and the rightmost occurrence of $T_0^{1,j-1}$ *(ii)* in $T_0^{1,j-1}(s_1 + \cdots + s_\ell)$ and $T_0^{1,j}$ and *(ii)* in the leftmost occurrence and middle occurrence of $T_0^{j+1,n}$. All other symbols are different because they are either in different columns, or in a row of different parity.
- Expression Ver-even$_{t,j}$: Each tile occurs at most three times. The only tiles that occur three times are the ones of the first j columns in odd rows. These tiles occur in $T_0^{1,n}$ in the beginning, in $T_0^{1,j-1}(s_1 + \cdots s_\ell)$ in the middle of the expression, and in the disjunction $(T_0^{1,j} + T'^{1,j})$ in the middle. The tiles that occur twice are the ones
  - in the last $n - j$ columns of odd rows: in the $T_0^{1,n}$ subexpression in the beginning and in the rightmost $T_0^{j+1,n}$ subexpression.
  - in the first $j-1$ columns of even rows: in the leftmost and the rightmost occurrence of $T_0^{1,j-1}$.
  - in the last $n-j$ columns of even rows: in the first and second occurrence of $T_0^{j+1,n}$, counting from left to right.
  - in the first j columns of the second-to-last row: in the subexpressions $T'^{1,j-1}(s'_1 + \cdots + s'_\ell)$ and $(T_0^{1,j} + T'^{1,j})$.

  All other tiles occur only once.
- Expression Ver-last$_{t,j}$: Each tile occurs at most twice. The only places where tiles occur more than once are *(i)* in the two $T'^{j+1,n}$ subexpressions, and *(ii)* in $T''^{j,j-1}(s''_1 + \cdots + s''_\ell)$ and $T''^{1,j}$.

This concludes the proof of (b). ∎

Corollary 3.11. intersection *is* pspace*-complete for one-unambiguous RE*$^{\leq 3}$ *expressions.*

A tractable fragment is the following:

Theorem 3.12. intersection *is in* ptime *for RE*(a, a^+).

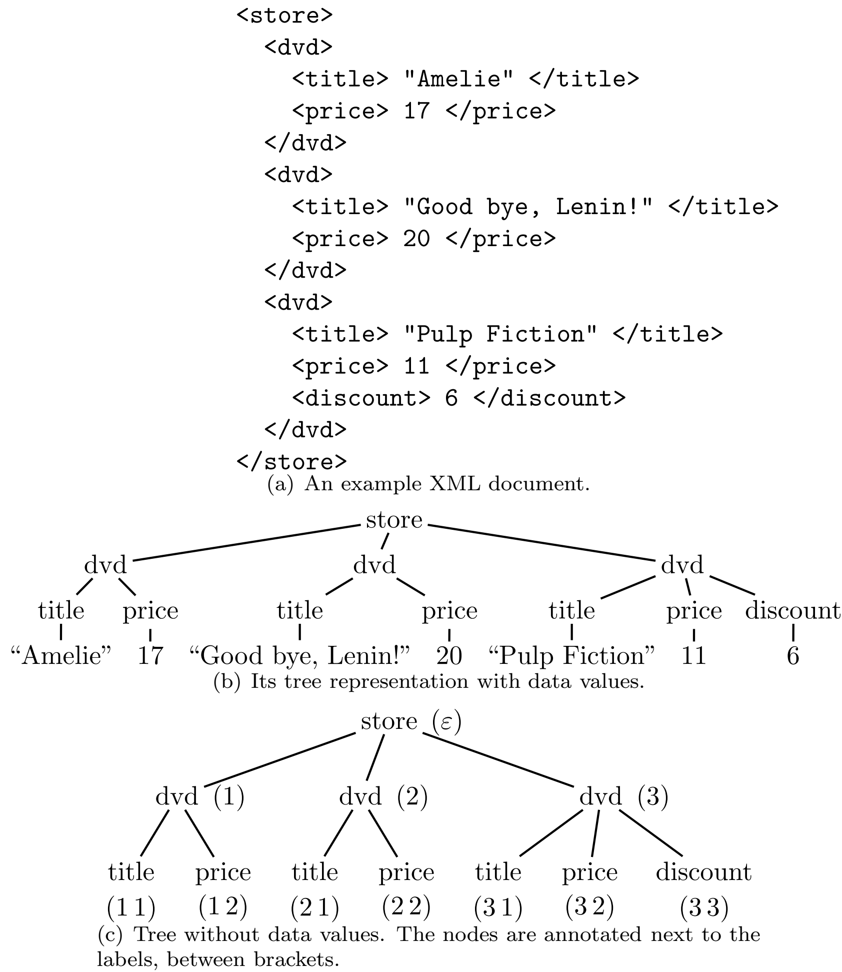
Proof. Suppose we are given RE(a, a^+) expressions $r_1, \ldots, r_n$ in sequence normal form. We describe a ptime method to decide non-emptiness of $L(r_1) \cap \cdots \cap L(r_n)$.

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 $a_j[k^i_j, l^i_j]$, for some k^i_j, l^i_j, where $k^i_j \geq 1$.

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

□

4. 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 reduce to the equivalence and inclusion problem for regular expressions. We also address the problem whether a set of schemas defines a common XML document. In the case of DTDs, the latter problem again reduces to the corresponding problem for regular expressions; in contrast, for XML Schema Definitions (XSDs) it does not, unless NP = EXPTIME.



```
<store>
  <dvd>
    <title> "Amelie" </title>
    <price> 17 </price>
  </dvd>
  <dvd>
    <title> "Good bye, Lenin!" </title>
    <price> 20 </price>
  </dvd>
  <dvd>
    <title> "Pulp Fiction" </title>
    <price> 11 </price>
    <discount> 6 </discount>
  </dvd>
</store>
```

(a) An example XML document.

(b) Its tree representation with data values.

(c) Tree without data values. The nodes are annotated next to the labels, between brackets.

FIG. 4.1. *An example of an XML document and its tree representations.*

4.1. Trees. It is common to view XML documents as finite trees with labels from a finite alphabet Σ. There is no limit on the number of children of a node. Figure 4.1 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.

We define the set of *unranked* Σ-*trees*, denoted by $\mathcal{T}_\Sigma$, as the smallest set of strings over Σ and the parenthesis symbols "(" and ")" such that, for $a \in \Sigma$ and $w \in \mathcal{T}_\Sigma^*$, $a(w)$ is in $\mathcal{T}_\Sigma$. So, a tree is either ε (empty) or is of the form $a(t_1 \cdots t_n)$ where each t_i is a tree. In the tree $a(t_1 \cdots t_n)$, the subtrees $t_1, \dots, t_n$ are attached to the root labeled a. We write a rather than $a()$. Notice that there is no a priori bound on the number of children of a node in a Σ-tree; such trees are therefore *unranked*. For every $t \in \mathcal{T}_\Sigma$, the *set of nodes* of t, denoted by $\mathrm{Dom}(t)$, is the set defined as follows:

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

Figure 4.1(c) contains a tree in which we annotated the nodes between brackets. Observe that the n child nodes of a node u are always $u1, \cdots, un$, from left to right. The *label* of a node u in the tree $t = a(t_1 \cdots t_n)$, denoted by $\mathrm{lab}^t(u)$, is defined as follows:

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

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

4.2. XML Schema Languages. We use extended context-free grammars and a restriction of extended DTDs [44] to abstract from DTDs and XML schemas.

DEFINITION 4.1. *Let $\mathcal{M}$ be a class of representations of regular string languages over Σ. A* DTD *over Σ is a tuple (Σ, d, s_d) where d is a function that maps Σ-symbols to elements of $\mathcal{M}$ and $s_d \in \Sigma$ is the* start symbol.

For convenience of notation, we often denote (Σ, d, s_d) by d and leave the alphabet and start symbol s_d implicit whenever this cannot give rise to confusion. A tree t *satisfies* d if *(i)* $\mathrm{lab}^t(\varepsilon) = s_d$ and, *(ii)* for every $u \in \mathrm{Dom}(t)$ with n children, $\mathrm{lab}^t(u1) \cdots \mathrm{lab}^t(un) \in L(d(\mathrm{lab}^t(u)))$. By $L(d)$ we denote the set of trees satisfying d.

For clarity, we write $a \to r$ 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:

$$\begin{array}{lcl} \text{store} & \to & \text{dvd dvd}^* \\ \text{dvd} & \to & \text{title price} \\ \text{title} & \to & \varepsilon \\ \text{price} & \to & \varepsilon. \end{array}$$

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 "dvd". Every "dvd"-labeled node has two children labeled "title" and "price", respectively.

We recall the definition of an extended DTD from [44]. The class of tree languages defined by EDTDs corresponds precisely to the regular (unranked) tree languages [11].

DEFINITION 4.2. *An* extended DTD *(EDTD) is a 5-tuple $E = (\Sigma, \Sigma^{type}, d, s_d, \mu)$, where Σ^{type} is an alphabet of* types, *(Σ^{type}, d, s_d) is a DTD over Σ^{type}, and μ is a mapping from Σ^{type} to Σ. Notice that μ can be extended to define a homomorphism*

on trees. A tree t then satisfies *an extended DTD if $t = \mu(t')$ for some $t' \in L(d)$. Again, we denote by $L(E)$ the set of trees satisfying E.*

In the sequel, we also denote by μ the homomorphic extension of μ to strings, trees, regular expressions, or classes of regular expressions. For ease of exposition, we always take $\Sigma^{\text{type}} = \{\mathsf{a}^i \mid 1 \leq i \leq k_a, a \in \Sigma, i \in \mathbf{N}\}$ for some natural numbers k_a, and we set $\mu(\mathsf{a}^i) = a$. For a node u in a Σ-tree t, we say that *a^i is a type of u with respect to E* when

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

For simplicity, we also denote EDTDs in examples in a way similar to DTDs, that is, we write $\mathsf{a}^i \to r$ rather than $d(\mathsf{a}^i) = r$.

Example 4.3. *A simple example of an EDTD is the following:*

$$\begin{array}{lcl}
\mathit{store}^1 & \to & (\mathit{dvd}^1 + \mathit{dvd}^2)^* \mathit{dvd}^2 (\mathit{dvd}^1 + \mathit{dvd}^2)^* \\
\mathit{dvd}^1 & \to & \mathit{title}^1 \ \mathit{price}^1 \\
\mathit{dvd}^2 & \to & \mathit{title}^1 \ \mathit{price}^1 \ \mathit{discount}^1 \\
\mathit{title}^1 & \to & \varepsilon \\
\mathit{price}^1 & \to & \varepsilon \\
\mathit{discount}^1 & \to & \varepsilon
\end{array}$$

The start symbol is store^1. 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 one DVD on sale. A tree in the language defined by this EDTD can be found in Figure 4.1(c). We refer to $(\mathit{dvd}^1 + \mathit{dvd}^2)^* \mathit{dvd}^2 (\mathit{dvd}^1 + \mathit{dvd}^2)^*$ as a typed expression and to $(\mathit{dvd} + \mathit{dvd})^* \mathit{dvd} (\mathit{dvd} + \mathit{dvd})^*$ as its untyped version.

The following restriction on EDTDs corresponds to the expressiveness of XML Schema [41, 34].

Definition 4.4. *A* single-type EDTD *($EDTD^{st}$) is an EDTD $(\Sigma, \Sigma^{type}, d, s_d, \mu)$ with the property that for every $\mathsf{a}^i \in \Sigma^{type}$, in the regular expression $d(\mathsf{a}^i)$ no two types b^j and b^k with $j \neq k$ occur.*

Example 4.5. *The EDTD of Example 4.3 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:*

$$\begin{array}{lcl}
\mathit{store}^1 & \to & \mathit{regulars}^1 \ \mathit{discounts}^1 \\
\mathit{regulars}^1 & \to & (\mathit{dvd}^1)^* \\
\mathit{discounts}^1 & \to & \mathit{dvd}^2 \ (\mathit{dvd}^2)^* \\
\mathit{dvd}^1 & \to & \mathit{title}^1 \ \mathit{price}^1 \\
\mathit{dvd}^2 & \to & \mathit{title}^1 \ \mathit{price}^1 \ \mathit{discount}^1 \\
\mathit{title}^1 & \to & \varepsilon \\
\mathit{price}^1 & \to & \varepsilon \\
\mathit{discount}^1 & \to & \varepsilon
\end{array}$$

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 defined by EDTDs 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 a^i of a node v w.r.t. the

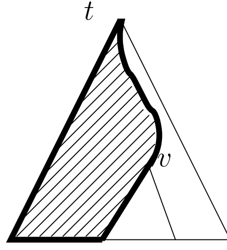


FIG. 4.2. *Illustration of the preceding of a node v in the tree t. In a one-pass preorder typeable EDTD E, the type a^i of v w.r.t. E is functionally determined by the striped area in t.*

EDTD is functionally determined by the *preceding* of v in t and the label of v itself. We illustrate the notion of the preceding of v in Figure 4.2. Here, the preceding of v is the striped area in t. For a more detailed discussion on the notion of one-pass preorder typing and its relation to restrained competition EDTDs defined next, we refer the reader to [34, 6].

DEFINITION 4.6. *Let $E = (\Sigma, \Sigma^{type}, d, s_d, \mu)$ be an EDTD. A regular expression r over Σ^{type}* restrains competition *if there are no strings $\mathsf{wa}^i\mathsf{v}$ and $\mathsf{wa}^j\mathsf{v}'$ in $L(r)$ with $i \neq j$. An EDTD E is* restrained competition *(EDTD^{rc}) if all regular expressions occurring in the definition of d restrain competition.*

Although the above definition seems superficially similar to the notion of one-unambiguous regular expressions, they are not equivalent. An expression is one-unambiguous if and only if its marking, seen as a typed expression, restrains competition. One-unambiguous regular expressions specify unique left-to-right matching of words without looking ahead, while restrained competition regular expressions specify the possibility to uniquely associate types in a left-to-right manner without looking ahead. For instance, in the example below, a dvd-symbol in a string matching $\mathsf{dvd}^*\mathsf{discounts}\ \mathsf{dvd}$ is assigned the type dvd^1 when it is not preceded by a discount-symbol and type dvd^2 otherwise. The expression corresponding to store in Example 4.3 does not restrain competition. Furthermore, when the untyped expression is one-unambiguous then clearly the typed extension restrains competition. The converse is not true: the expression $(\mathsf{a}^1 + \mathsf{b}^2)^*\mathsf{a}^1$ is not one-unambiguous but restrains competition. This is discussed in more detail in [34].

EXAMPLE 4.7. *An example of a restrained competition EDTD that is not single-type is given next:*

$$
\begin{array}{lcl}
\mathit{store}^1 & \to & (\mathit{dvd}^1)^*\ \mathit{discounts}^1\ (\mathit{dvd}^2)^* \\
\mathit{discounts}^1 & \to & \varepsilon \\
\mathit{dvd}^1 & \to & \mathit{title}^1\ \mathit{price}^1 \\
\mathit{dvd}^2 & \to & \mathit{title}^1\ \mathit{price}^1\ \mathit{discount}^1 \\
\mathit{title}^1 & \to & \varepsilon \\
\mathit{price}^1 & \to & \varepsilon \\
\mathit{discount}^1 & \to & \varepsilon
\end{array}
$$

The start symbol is store^1.

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: DTD $\subsetneq$ EDTD$^{\mathrm{st}}$ $\subsetneq$ EDTD$^{\mathrm{rc}}$ $\subsetneq$ EDTD [41].

REMARK 4.8. Murata et al. observed that there is a straightforward deterministic algorithm to check validity of a tree t with respect to an EDTD$^{\text{st}}$ or EDTD$^{\text{rc}}$ E [41, 34]. It proceeds top-down and assigns to every node with some symbol a a type a^i. To the root the start symbol of d is assigned; then, for every interior node u with type a^i, it is checked whether the children of u match $\mu(d(\mathsf{a}^i))$; if not, the tree is rejected; otherwise, as E is 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 (Σ, d, s_d) 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 (Σ, d, a) where $d(a) = a$ is not reduced because there exists no finite tree satisfying the DTD. We say that an EDTD $(\Sigma, \Sigma^{\text{type}}, d, s_d, \mu)$ is *reduced* if d is reduced.

We can reduce a DTD (Σ, d, s_d) in linear time in four steps:

(1) Compute in a bottom-up manner the set of all symbols $S = \{a \mid L((\Sigma, d, a)) \neq \emptyset\}$.
(2) Replace every occurrence of a symbol in $\Sigma \setminus S$ in all regular expressions in the definition of d by $\emptyset$, and remove all symbols in $\Sigma \setminus S$ from the DTD.
(3) In the resulting DTD, compute the set $R \subseteq (\Sigma \setminus S)$ of symbols that are *reachable* from the start symbol s_d. Here, a symbol a is reachable from s_d if there is a string $w_1 a w_2 \in L(d(b))$ and b is reachable from s_d. Furthermore, s_d is reachable from s_d.
(4) Remove all symbols in $S \setminus R$ in all regular expressions in the definition of d, and remove their respective regular expressions.

We note that this algorithm is not new. It appears as Theorem 1 in [3]. It is easy to see that the resulting DTD $(S \cap R, d', s_d)$ is reduced and that $L(d) = L(d')$. Concerning the time complexity of the algorithm, Steps (2) and (4) are obviously in linear time. Step (1) is known to be in linear time since testing whether a symbol in a context-free grammar can produce a non-empty string of terminals. It is shown in [3] that the algorithm for standard context-free grammars can be lifted to extended ones. Step (3) can be carried out in linear time by a straightforward reachability algorithm. Hence, the overall algorithm can also be carried out in linear time. 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 4.9. *Let $\mathcal{M}$ be a subclass of the class of DTDs, EDTDs, EDTDsts or EDTD^{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 $\bigcap_{i=1}^n L(d_i) \neq \emptyset$?*

4.3. Inclusion and Equivalence of XML Schema Languages. We show that the complexity for inclusion and equivalence of DTDs, EDTD$^{\text{st}}$ s, and EDTD$^{\text{rc}}$s reduces to the corresponding problem on regular expressions. For a class $\mathcal{R}$ of regular expressions over $\Sigma \uplus \Sigma^{\text{type}}$, we denote by DTD($\mathcal{R}$), EDTD($\mathcal{R}$), EDTD$^{\text{st}}$($\mathcal{R}$), and EDTD$^{\text{rc}}$($\mathcal{R}$), the class of DTDs, EDTDs, EDTD$^{\text{st}}$ and EDTD$^{\text{rc}}$s with regular expressions in $\mathcal{R}$. By $\mathcal{R}^{\text{type}}$ and $\mathcal{R}^{\text{label}}$ we denote the expressions in $\mathcal{R}$ restricted to Σ^{type} and Σ, respectively.

We call a complexity class $\mathcal{C}$ *closed under positive reductions* if the following holds for every $O \in \mathcal{C}$. Let L' be accepted by a deterministic polynomial-time Turing machine M with oracle O (denoted $L' = L(M^O)$). Let M further have the property that

$L(M^A) \subseteq L(M^B)$ whenever $A \subseteq B$. Then L' is also in $\mathcal{C}$. For a more precise definition of this notion we refer the reader to [23]. 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 every Σ-symbol a, S_a be the set $\{a\} \cup \{\mathsf{a}^i \mid i \in \mathbf{N}\}$. We say that a homomorphism h on strings is *label-preserving* if $h(S_a) \subseteq S_a$ for every Σ-symbol a. We call a class of regular expressions $\mathcal{R}$ *closed under label-preserving homomorphisms* if, for every $r \in \mathcal{R}$ and every label-preserving homomorphism h, $h(r) \in \mathcal{R}$. For our purposes, it is important to note that CHAREs, and their restricted fragments we study in this article, and one-unambiguous regular expressions are closed under label-preserving homomorphisms. For $\mathrm{RE}^{\leq k}$, we say that a typed expression is in $\mathrm{RE}^{\leq k}$ whenever its untyped version is. So, $\mathrm{RE}^{\leq k}$ is closed under label-preserving homomorphisms.

We now show that deciding INCLUSION or EQUIVALENCE for DTDs, $\mathrm{EDTD}^{\mathrm{st}}$s, or $\mathrm{EDTD}^{\mathrm{rc}}$s is essentially not harder than deciding INCLUSION or EQUIVALENCE for the regular expressions that they use. The overall scheme of the proof of the following theorem is to construct a polynomial time deterministic Turing Machine that solves the problem for the DTD class under consideration which uses the corresponding problem for regular expressions as an oracle.

THEOREM 4.10. *Let $\mathcal{R}$ be a class of regular expressions which is closed under label-preserving homomorphisms. Let $\mathcal{R}^{label}$ and $\mathcal{R}^{type}$ be as defined above. Let $\mathcal{C}$ be a complexity class which is closed under positive reductions. Then the following are equivalent:*
(a) INCLUSION *for $\mathcal{R}^{label}$-expressions is in $\mathcal{C}$.*
(b) INCLUSION *for DTD($\mathcal{R}^{label}$) is in $\mathcal{C}$.*
(c) INCLUSION *for EDTDst($\mathcal{R}^{type}$) is in $\mathcal{C}$.*
(d) INCLUSION *for EDTD^{rc}($\mathcal{R}^{type}$) is in $\mathcal{C}$.*
The corresponding statements hold for EQUIVALENCE.

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

For the implication from (c) to (b), let (Σ, d_1, s_1) and (Σ, d_2, s_2) be two DTDs in $\mathrm{DTD}(\mathcal{R}^{\mathrm{label}})$. Let, for every $i = 1, 2$, d_i' be the DTD obtained from d_i by replacing every Σ-symbol a with $\nu(a)$ and every regular expression $d_i(a)$ by $\nu(d_i(a))$, where ν is the label-preserving homomorphism mapping every Σ-symbol b to the type b^1. As $\mathcal{R}$ is closed under label-preserving homomorphisms, the $\mathrm{EDTD}^{\mathrm{st}}$s $E_1 = (\Sigma, \Sigma^{\mathrm{type}}, d_1', \mathsf{s}_1^1, \mu)$ and $E_2 = (\Sigma, \Sigma^{\mathrm{type}}, d_2', \mathsf{s}_2^1, \mu)$ are in $\mathrm{EDTD}^{\mathrm{st}}(\mathcal{R}^{\mathrm{type}})$. 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 $E_1 = (\Sigma, \Sigma_1^{\mathrm{type}}, d_1, \mathsf{s}_1, \mu_1)$ and $E_2 = (\Sigma, \Sigma_2^{\mathrm{type}}, d_2, \mathsf{s}_2, \mu_2)$ be two reduced $\mathrm{EDTD}^{\mathrm{rc}}(\mathcal{R}^{\mathrm{type}})$s. We define a correspondence relation $R \subseteq \Sigma_1^{\mathrm{type}} \times \Sigma_2^{\mathrm{type}}$ as follows:

(1) $(\mathsf{s}_1, \mathsf{s}_2) \in R$; and,
(2) if $(\mathsf{a}^i, \mathsf{a}^j) \in R$, $\mathsf{w}_1\mathsf{b}^k\mathsf{v}_1 \in L(d_1(\mathsf{a}^i))$, $\mathsf{w}_2\mathsf{b}^\ell\mathsf{v}_2 \in L(d_2(\mathsf{a}^j))$ and $\mu_1(\mathsf{w}_1) = \mu_2(\mathsf{w}_2)$ then $(\mathsf{b}^k, \mathsf{b}^\ell) \in R$.

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 4.11. *A pair $(\mathsf{a}^i, \mathsf{a}^j)$ 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 4.8 assigns type a^i to u with respect to E_1; and

(2) the algorithm of Remark 4.8 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 PTIME in a top-down left-to-right manner. To this end, let $A_{\mathsf{a}^i} = (Q_{\mathsf{a}^i}, \Sigma_1^{\text{type}}, \delta_{\mathsf{a}^i}, I_{\mathsf{a}^i}, F_{\mathsf{a}^i})$ and $A_{\mathsf{a}^j} = (Q_{\mathsf{a}^j}, \Sigma_2^{\text{type}}, \delta_{\mathsf{a}^j}, I_{\mathsf{a}^j}, F_{\mathsf{a}^j})$ be the Glushkov-automata of $d_1(\mathsf{a}^i)$ and $d_2(\mathsf{a}^j)$, respectively (see [22, 12]). We now give an NLOGSPACE decision procedure that tests, given $(\mathsf{a}^i, \mathsf{a}^j) \in R$, whether there are $\mathsf{w}_1\mathsf{b}^k\mathsf{v}_1 \in L(d_1(\mathsf{a}^i))$ and $\mathsf{w}_2\mathsf{b}^\ell\mathsf{v}_2 \in L(d_2(\mathsf{a}^j))$ with $\mu_1(\mathsf{w}_1) = \mu_2(\mathsf{w}_2)$. The algorithm guesses the strings $\mathsf{w}_1\mathsf{b}^k$ and $\mathsf{w}_2\mathsf{b}^\ell$ one symbol at a time, while simulating A_{a^i} on $\mathsf{w}_1\mathsf{b}^k$ and A_{a^j} on $\mathsf{w}_2\mathsf{b}^\ell$. 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 $\mathsf{w}_1\mathsf{b}^k$ and a symbol x_2 of $\mathsf{w}_2\mathsf{b}^\ell$, we verify whether $\mu_1(\mathsf{x}_1) = \mu_2(\mathsf{x}_2)$ and we overwrite the current states $q_1 \in Q_{\mathsf{a}^i}$ and $q_2 \in Q_{\mathsf{a}^j}$ by a state in $\delta_{\mathsf{a}^i}(q_1, \mathsf{x}_1)$ and $\delta_{\mathsf{a}^j}(q_2, \mathsf{x}_2)$, respectively. We nondeterministically determine when we stop guessing the strings $\mathsf{w}_1\mathsf{b}^k$ and $\mathsf{w}_2\mathsf{b}^\ell$. 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 and q_2, which can also be decided in 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 EDTD$^{\text{rc}}$s reduces to testing INCLUSION and EQUIVALENCE for the regular expressions. This follows from Claim 4.12 below and the closure property of $\mathcal{C}$. The statement for EQUIVALENCE follows likewise.

CLAIM 4.12. *With the notation as above, $L(E_1)$ is included in (resp., equal to) $L(E_2)$ if and only if, for every $\mathsf{a}^i \in \Sigma_1^{type}$ and $\mathsf{a}^j \in \Sigma_2^{type}$ with $(\mathsf{a}^i, \mathsf{a}^j) \in R$, the regular expression $\mu_1(d_1(\mathsf{a}^i))$ is included in (resp., equivalent to) $\mu_2(d_2(\mathsf{a}^j))$.*

Proof. We give a proof for INCLUSION. EQUIVALENCE then immediately follows. We can assume without loss of generality that $\mu_1(\mathsf{s}_1) = \mu_2(\mathsf{s}_2)$.

($\Rightarrow$) We prove this direction by contraposition. Suppose that there is a pair $(\mathsf{a}^i, \mathsf{a}^j) \in R$ for which the regular expression $\mu_1(d_1(\mathsf{a}^i))$ is not included in $\mu_2(d_2(\mathsf{a}^j))$. We then need to show that $L(E_1)$ is not included in $L(E_2)$.

Thereto, let w be a counterexample Σ-string in $L(\mu_1(d_1(\mathsf{a}^i))) - L(\mu_2(d_2(\mathsf{a}^j)))$. From Observation 4.11, 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 $\mu_2(d_2(\mathsf{a}^j))$. 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 $\mathsf{a}^i \in \Sigma_1^{\text{type}}$ and $\mathsf{a}^j \in \Sigma_2^{\text{type}}$, with $(\mathsf{a}^i, \mathsf{a}^j) \in R$, such that $\mu_1(d_1(\mathsf{a}^i))$ is not included in $\mu_2(d_2(\mathsf{a}^j))$.

As $t \notin L(E_2)$, there is a node to which the algorithm in Remark 4.8 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 $\mu_2(d_2(\mathsf{a}^j))$. 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 4.8 assigns to u with respect to E_1. So, by Observation 4.11, we have that $(\mathsf{a}^i, \mathsf{a}^j) \in R$. Let w be the concatenation of the labels of u's children. But as $w \in L(\mu_1(d_1(\mathsf{a}^i)))$, and w is not in $L(\mu_2(d_2(\mathsf{a}^j)))$, we have that $\mu_1(d_1(\mathsf{a}^i))$ is not

included in $\mu_2(d_2(\mathsf{a}^j))$. ☐

This concludes the proof of Theorem 4.10 ☐

Although Theorem 4.10 is only stated for a single class $\mathcal{R}$ of regular expressions, the proof of the theorem analogously allows to carry over the complexity of inclusion testing between regular expressions from two different classes of regular expressions $\mathcal{R}_\infty$ and $\mathcal{R}_\in$ to the complexity of inclusion testing between two schemas E_1 and E_2 such that E_1 only uses expressions from $\mathcal{R}_\infty$ and E_2 only from $\mathcal{R}_\in$.

4.4. 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 4.14, 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 4.15. Theorem 4.15 shows that there is a class of EDTD$^{\mathrm{st}}$s for which the intersection problem is EXPTIME-hard. As the intersection problem for the regular expressions we use in the proof is in NP, the intersection problem for single-type or restrained competition EDTDs cannot be reduced to the corresponding problem for regular expressions, unless NP = EXPTIME.

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 *generalized intersection* problem for $\mathcal{R}$ is to determine, given an arbitrary number of expressions $r_1, \dots, r_n \in \mathcal{R}$ and a set $S \subseteq \Sigma$, whether $\bigcap_{i=1}^{n} L(r_i) \cap S^* \neq \emptyset$.

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

LEMMA 4.13. *For each class of regular expressions $\mathcal{R}$ and complexity class $\mathcal{C}$ 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. The direction from (b) to (a) is trivial. We prove the direction from (a) to (b). Let $r_1, \dots, r_n$ 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, $\bigcap_{i=1}^{n} L(r_i') \neq \emptyset$ if and only if $\bigcap_{i=1}^{n} L(r_i) \cap S^* \neq \emptyset$. ☐

We are now ready to show the theorem for DTDs.

THEOREM 4.14. *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 $d_1, \dots, d_n$ be in DTD($\mathcal{R}$). We assume without loss of generality that $d_1, \dots, d_n$ all have the same start symbol. We compute, for each $1 \leq i \leq |\Sigma| + 1$, the set S_i of symbols with the following property:

$$a \in S_i \text{ if and only if there is a tree of depth at most } i \text{ in } L((\Sigma, d_1, a)) \cap \cdots \cap L((\Sigma, d_n, a)). \quad (*)$$

Notice that we have replaced the start symbols of $d_1, \dots, d_n$ with a in the above definition. Initially, we set $S_1 = \{a \mid \varepsilon \in L(d_j(a)) \text{ for all } j \in \{1, \dots, n\}\}$. For every

$i > 1$, S_i is the set S_{i-1} extended with all symbols a for which $S_{i-1}^* \cap L(d_1(a)) \cap \cdots \cap L(d_n(a)) \neq \emptyset$. Clearly, $S_{k+1} = S_k$ for $k := |\Sigma|$. It can be shown by a straightforward induction on i that $(*)$ 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 ε. So, S_1 can be computed in PTIME and hence in $\mathcal{C}$. From Lemma 4.13, 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 4.14. ∎

The following theorem shows that single-type and restrained competition EDTDs probably cannot be added to Theorem 4.14. Indeed, by Theorem 3.7, the intersection problem for $\text{RE}((+a), w?, (+a)?, (+a)^*)$ expressions is in NP and by Theorem 4.15, the intersection problem is already EXPTIME-complete for single-type EDTDs with $\text{RE}((+a), w?, (+a)?, (+a)^*)$ expressions. The proof of Theorem 4.15 is similar to the proof that intersection of deterministic top-down tree automata is EXPTIME-complete [48]. 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 defined by the EDTD consisting of the rules $a^1 \rightarrow b^1c^1 + c^1b^1$, $b^1 \rightarrow \varepsilon$, $c^1 \rightarrow \varepsilon$. 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 4.15. INTERSECTION *for* $\textit{EDTD}^{st}((+a), w?, (+a)?, (+a)^*)$ *is* EXPTIME-*complete.*

Proof. The upper bound can be obtained by a straightforward logspace reduction to the INTERSECTION problem for non-deterministic tree automata, which is in EXPTIME.

For the lower bound, 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 Σ we take $T \uplus \{\#\}$. We define E_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 every possible answer 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 rows $\bar{b}$ and $\bar{t}$ will not be present in the tree. The start symbol of E_0 is #. The EDTD E_0 uses the alphabet $\Sigma^{\text{type}} = \{\#, \texttt{error}, \texttt{t}_1^1, \ldots, \texttt{t}_k^1, \texttt{t}_1^2, \ldots, \texttt{t}_k^2\}$. The rules are as follows:

- $\# \rightarrow (\texttt{t}_1^1 + \cdots + \texttt{t}_k^1)$;
- for every $t \in T$ and $1 \leq i \leq k$, $\texttt{t}_i^1 \rightarrow (\texttt{t}_1^2 \cdots \texttt{t}_k^2)?$; and
- for every $t \in T$ and $1 \leq i \leq k$, $\texttt{t}_i^2 \rightarrow (\texttt{t}_1^1 + \cdots + \texttt{t}_k^1 + \texttt{error})?$.

Here, a tile with label t_i is assigned the type $\texttt{t}_i^1$ (respectively $\texttt{t}_i^2$) 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.

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

To simplify the exposition below, we start by making the following observation. Essentially, we argue that a single-type EDTD can test whether a regular condition

holds on all paths of a tree. More precisely, given a DFA M such that each string accepted by M starts with the same symbol, there exists a single-type EDTD E_M that defines precisely the trees for which each sequence of labels from the root to a leaf is accepted by M. Furthermore, this EDTD can be constructed from M using only logarithmic space. To make this more concrete, let M be an arbitrary DFA accepting strings over alphabet $\Sigma = T \uplus \{\#\}$. We can assume without loss of generality that M's state set is $\{0, \ldots, m\}$ for some $m \in \mathbf{N}$, that is, $M = (Q_M, \Sigma, \delta_M, \{0\}, F_M)$ with state set $Q = \{0, \ldots, m\}$. Furthermore, let M have the property that there is an $a \in \Sigma$ such that every string in $L(M)$ starts with a. Then, the single-type EDTD E_M can be constructed as follows. Intuitively, we will use the types of E_M to remember the current state of M. That is, E_M will assign type b^i to a node x in a tree if it is labeled with b and M arrives in state i when having read the sequence of Σ-symbols on path from the root to x. Formally, $E_M = (\Sigma, \Sigma^{\text{type}}, d, \mathsf{a}^l, \mu)$, where $\Sigma = T \uplus \{\#\}$, $\Sigma^{\text{type}} = \{b^i \mid b \in \Sigma, i \in Q_M\}$, and $l \in Q$ is the unique state such that $\delta_M(0, a) = l$. Hence, for the root node labeled a, E_M assigns the type a^l such that M is in state l after having read a. It is now easy to generalize this. Indeed, for every $i, j \in Q$ and $b \in \Sigma$ for which $\delta_M(i, b) = \{j\}$, d contains the rule *(i)* $\mathsf{b}^i \to (\mathsf{t}_1^j + \cdots + \mathsf{t}_k^j + \#^j)^*$ if $i \in F$, and *(ii)* $\mathsf{b}^i \to (\mathsf{t}_1^j + \cdots + \mathsf{t}_k^j + \#^j)^+$, otherwise. In case *(i)*, when $i \in F$, then the path from the root to the b-labeled node is accepted by M. Hence, the label b is allowed to be a leaf. In case *(ii)*, when $i \notin F$, then the path to the b-labeled node is not accepted by M. Hence, b is not allowed to be a leaf. This concludes our observation.

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)* its length 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 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 ith tile of $\bar{b}$ and the ith 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 ith tile of the last row and the ith tile of $\bar{t}$ should satisfy the vertical constraints. This can be checked by one automaton.

Clearly, $L(E_0) \cap L(E_{M_1}) \cap \cdots \cap L(E_{M_\ell})$ is non-empty if and only if player CONSTRUCTOR has a winning strategy. ∎

5. Conclusion. We addressed the complexity of the basic decision problems for a very simple fragment of regular expressions: chain regular expressions. Surprisingly, inclusion of $\text{RE}(a, a^*)$ and $\text{RE}(a, (+a)^*)$ is already hard for CONP and PSPACE respectively, in contrast to inclusion for $\text{RE}(a, \Sigma, \Sigma^*)$, which is in PTIME. We left the following complexities open: *(i)* intersection of $\text{RE}(a, w^+)$ expressions; and, equivalence for the general class of CHAREs or any fragment extending $\text{RE}(a, a^*)$ or $\text{RE}(a, a?)$. Moreover, we showed that 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.

Acknowledgment. We are grateful to Sebastian Bala for referring us to his work [4], which contains the proof ideas that are needed for Theorem 3.7(c). Furthermore, we are grateful to the reviewers of the full version this paper, who provided us with many useful suggestions and comments.

REFERENCES

[1] P. A. Abdulla, A. Collomb-Annichini, A. Bouajjani, and B. Jonsson. Using forward reachability analysis for verification of lossy channel systems. *Formal Methods in System Design*, 25(1):39–65, 2004.

[2] S. Abiteboul, P. Buneman, and D. Suciu. *Data on the Web : From Relations to Semistructured Data and XML*. Morgan Kaufmann, San Francisco, 1999.

[3] J. Albert, D. Giammerresi, and D. Wood. Normal form algorithms for extended context free grammars. *Theoretical Computer Science*, 267(1–2):35–47, 2001.

[4] S. Bala. Intersection of regular languages and star hierarchy. In *Proceedings of the 29th International Colloquium on Automata, Languages and Programming (ICALP)*, pages 159–169, Berlin, 2002. Springer.

[5] M. Benedikt, W. Fan, and F. Geerts. Xpath satisfiability in the presence of DTDs. *Journal of the ACM*, 55(2), 2008.

[6] G. J. Bex, W. Martens, F. Neven, and T. Schwentick. Expressiveness of XSDs: from practice to theory, there and back again. In *Proceedings of the 14th International Conference on World Wide Web (WWW)*, pages 712–721, USA, 2005. ACM.

[7] G. J. Bex, F. Neven, T. Schwentick, and K. Tuyls. Inference of concise DTDs from XML data. In *Proceedings of the 32nd International Conference on Very Large Data Bases (VLDB)*, pages 115–126, USA, 2006. ACM.

[8] G.J. Bex, F. Neven, and J. Van den Bussche. DTDs versus XML schema: A practical study. In *Proceedings of the 7th International Workshop on the Web and Databases (WebDB)*, pages 79–84, 2004.

[9] T. Bray, J. Paoli, C.M. Sperberg-McQueen, E. Maler, and F. Yergeau. Extensible Markup Language (XML). Technical report, World Wide Web Consortium, February 2004. http://www.w3.org/TR/REC-xml/.

[10] A. Brüggemann-Klein and Wood D. One unambiguous regular languages. *Information and Computation*, 140(2):229–253, 1998.

[11] A. Brüggemann-Klein, M. Murata, and D. Wood. Regular tree and regular hedge languages over unranked alphabets. Technical Report HKUST-TCSC-2001-0, The Hongkong University of Science and Technology, 2001.

[12] A. Brüggemann-Klein and D. Wood. One-unambiguous regular languages. *Information and Computation*, 142(2):182–206, 1998.

[13] A. Brüggemann-Klein and D. Wood. Caterpillars: A context specification technique. *Markup Languages*, 2(1):81–106, 2000.

[14] D. Calvanese, De G. Giacomo, M. Lenzerini, and M. Y. Vardi. Reasoning on regular path queries. *SIGMOD Record*, 32(4):83–92, 2003.

[15] B. S. Chlebus. Domino-tiling games. *Journal of Computer and System Sciences*, 32(3):374–392, 1986.

[16] B. Choi. What are real DTDs like? In *Proceedings of the 5th International Workshop on the Web and Databases (WebDB)*, pages 43–48, 2002.

[17] A. Deutsch, M. F. Fernandez, and D. Suciu. Storing semistructured data with STORED. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 431–442, USA, 1999. ACM.

[18] M.R. Garey and D.S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W.H. Freeman, USA, 1979.

[19] W. Gelade, M. Gyssens, and W. Martens. Regular expressions with counting: Weak versus strong determinism. In *Proceedings of the 33rd International Symposium on Mathematical Foundations of Computer Science (MFCS)*, 2009. To appear.

[20] W. Gelade, W. Martens, and F. Neven. Optimizing schema languages for XML: Numerical constraints and interleaving. *SIAM Journal on Computing*, 38(5):2021–2043, 2009.

[21] G. Ghelli, D. Colazzo, and C. Sartiani. Efficient inclusion for a class of XML types with interleaving and counting. 11th International Symposium on Database Programming Languages (DBPL), 2007.

[22] V. Glushkov. Abstract theory of automata. *Uspekhi Matematicheskikh Nauk*, 16(1):3–62, 1961.

English translation in Russian Mathematical Surveys, vol. 16, pp. 1–53, 1961.

[23] L. Hemaspaandra and M. Ogihara. *The Complexity Theory Companion*. Springer, Berlin, 2002.

[24] H. Hosoya and B. C. Pierce. XDuce: A statically typed XML processing language. *ACM Transactions on Internet Technology*, 3(2):117–148, 2003.

[25] H. B. Hunt III, D. J. Rosenkrantz, and T. G. Szymanski. On the equivalence, containment, and covering problems for the regular and context-free languages. *Journal of Computer and System Sciences*, 12(2):222–268, 1976.

[26] P. Kilpeläinen and R. Tuhkanen. One-unambiguity of regular expressions with numeric occurrence indicators. *Information and Computation*, 205(6):890–916, 2007.

[27] C. Koch, S. Scherzinger, N. Schweikardt, and Bernhard Stegmaier. Schema-based scheduling of event processors and buffer minimization for queries on structured data streams. In *Proceedings of the 30th International Conference on Very Large Data Bases (VLDB)*, pages 228–239, San Francisco, 2004. Morgan Kaufmann.

[28] D. Kozen. Lower bounds for natural proof systems. In *Proceedings of the 18th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 254–266, USA, 1977. IEEE.

[29] L. Libkin. Logics for unranked trees: An overview. *Logical Methods in Computer Science*, 2(3), 2006.

[30] M. Mani. Keeping chess alive - Do we need 1-unambiguous content models? In *Extreme Markup Languages*, Montreal, Canada, 2001.

[31] I. Manolescu, D. Florescu, and D. Kossmann. Answering XML Queries on Heterogeneous Data Sources. In *Proceedings of the 27th International Conference on Very Large Data Bases (VLDB)*, pages 241–250, San Francisco, 2001. Morgan Kaufmann.

[32] W. Martens and F. Neven. On the complexity of typechecking top-down XML transformations. *Theoretical Computer Science*, 336(1):153–180, 2005.

[33] W. Martens, F. Neven, and T. Schwentick. Complexity of decision problems for simple regular expressions. In *Proceedings of the 29th Symposium on Mathematical Foundations of Computer Science (MFCS)*, pages 889–900, Berlin, 2004. Springer.

[34] W. Martens, F. Neven, T. Schwentick, and G.J. Bex. Expressiveness and complexity of XML schema. *ACM Transactions on Database Systems*, 31(3):770–813, 2006.

[35] M. Marx. XPath with conditional axis relations. In *Proceedings of the 9th International Conference on Extending Database Technology (EDBT)*, pages 477–494, Berlin, 2004. Springer.

[36] G. Miklau and D. Suciu. Containment and equivalence for a fragment of XPath. *Journal of the ACM*, 51(1):2–45, 2004.

[37] T. Milo and D. Suciu. Index structures for path expressions. In *Proceedings of the 7th International Conference on Database Theory (ICDT)*, pages 277–295, Berlin, 1999. Springer.

[38] T. Milo, D. Suciu, and V. Vianu. Typechecking for XML transformers. *Journal of Computer and System Sciences*, 66(1):66–97, 2003.

[39] D. Mount. *Bioinformatics: Sequence and Genome Analysis*. Cold Spring Harbor Laboratory Press, 2001.

[40] M. Murata. Relax. `http://www.xml.gr.jp/relax/`.

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

[42] F. Neven. Automata, logic, and XML. In *Proceedings of the 16th Conference for Computer Science Logic (CSL 2002)*, pages 2–26, Berlin, 2002. Springer.

[43] F. Neven and T. Schwentick. On the complexity of XPath containment in the presence of disjunction, DTDs, and variables. *Logical Methods in Computer Science*, 2(3), 2006.

[44] Y. Papakonstantinou and V. Vianu. DTD inference for views of XML data. In *Proceedings of the 19th Symposium on Principles of Database Systems (PODS)*, pages 35–46, USA, 2000. ACM Press.

[45] T. Schwentick. Automata for XML — a survey. *J. Comput. Syst. Sci.*, 73(3):289–315, 2007.

[46] R. Sedgewick. *Algorithms*. Addison-Wesley, Reading, MA, 1983.

[47] H. Seidl. Deciding equivalence of finite tree automata. *SIAM Journal on Computing*, 19(3):424–437, 1990.

[48] H. Seidl. Haskell overloading is DEXPTIME-complete. *Information Processing Letters*, 52(2):57–60, 1994.

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

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

[51] L. J. Stockmeyer and A. R. Meyer. Word problems requiring exponential time: Preliminary report. In *Conference Record of the 5th Annual ACM Symposium on Theory of Computing (STOC)*, pages 1–9, USA, 1973. ACM.

[52] B. ten Cate. The expressivity of XPath with transitive closure. In *Proceedings of the 25th Symposium on Principles of Database Systems (PODS)*, pages 328–337, 2006.
[53] E. van der Vlist. *XML Schema*. O'Reilly, 2002.
[54] E. van der Vlist. *Relax NG*. O'Reilly, 2003.
[55] V. Vianu. A Web odyssey: from Codd to XML. *SIGMOD Record*, 32(2):68–77, 2003.
[56] G. Wang, M. Liu, J. Xu Yu, B. Sun, G. Yu, J. Lv, and H. Lu. Effective schema-based XML query optimization techniques. In *International Database Engineering and Applications Symposium (IDEAS)*, pages 230–235, 2003.