Optimizing Schema Languages for XML: Numerical Constraints and Interleaving

Wouter Gelade*, Wim Martens, and Frank Neven

Hasselt University and Transnational University of Limburg
School for Information Technology
{firstname.lastname}@uhasselt.be

Abstract. The presence of a schema offers many advantages in processing, translating, querying, and storage of XML data. Basic decision problems like equivalence, inclusion, and non-emptiness of intersection of schemas form the basic building blocks for schema optimization and integration, and algorithms for static analysis of transformations. It is thereby paramount to establish the exact complexity of these problems. Most common schema languages for XML can be adequately modeled by some kind of grammar with regular expressions at right-hand sides. In this paper, we observe that apart from the usual regular operators of union, concatenation and Kleene-star, schema languages also allow numerical occurrence constraints and interleaving operators. Although the expressiveness of these operators remain within the regular languages, their presence or absence has significant impact on the complexity of the basic decision problems. We present a complete overview of the complexity of the basic decision problems for DTDs, XSDs and Relax NG with regular expressions incorporating numerical occurrence constraints and interleaving. We also discuss chain regular expressions and the complexity of the schema simplification problem incorporating the new operators.

1 Introduction

XML is the lingua franca for data exchange on the Internet [1]. Within applications or communities, XML data is usually not arbitrary but adheres to some structure imposed by a schema. The presence of such a schema not only provides users with a global view on the anatomy of the data, but far more importantly, it enables automation and optimization of standard tasks like (*i*) searching, integration, and processing of XML data (cf., e.g., [11, 22, 25, 42]); and, (*ii*) static analysis of transformations (cf., e.g., [2, 16, 26, 32]). Decision problems like equivalence, inclusion and non-emptiness of intersection of schemas, hereafter referred to as *the basic decision problems*, constitute essential building blocks in solutions for the just mentioned optimization and static analysis problems. Additionally, the basic decision problems are fundamental for schema minimization (cf., e.g.,

* Research Assistant of the Fund for Scientific Research - Flanders (Belgium)

shop → regular* & discount-box*
regular → cd
discount-box → cd[10,12] price
cd → artist & title & price

Fig. 1. A sample schema using the numerical occurrence and interleave operators. The schema defines a shop that sells CDs and offers a special price for boxes of 10–12 CDs.

[9, 29]). Because of their widespread applicability, it is therefore important to establish the exact complexity of the basic decision problems for the various XML schema languages.

The most common schema languages for XML are DTD, XML Schema [38], and Relax NG [8] and can be modeled by grammar formalisms [31]. 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) [33] or equivalently, unranked tree automata [6], 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 [28]. In fact, they can be abstracted by single-type EDTDs. As detailed in [27], the relationship between schema formalisms and grammars provides direct upper and lower bounds for the complexity of the basic decision problems.

A closer inspection of the various schema specifications reveals that the above abstractions in terms of grammars with regular expressions is too coarse. Indeed, in addition to the conventional regular expression operators like concatenation, union, and Kleene-star, the XML Schema and the Relax NG specification allow two other operators as well:

(1) Both the XML Schema and the Relax NG specification allow a certain form of unordered concatenation: the `ALL` and the `interleave` operator, respectively. This operator is actually the resurrection of the &-operator from SGML DTDs that was excluded from the definition of XML DTDs. Although there are restrictions on the use of `ALL` and `interleave`, we consider the operator in its unrestricted form. We refer by RE(&) to such regular expressions with the unordered concatenation operator.
(2) The XML Schema specification allows to express numerical occurrence constraints which define the minimal and maximal number of times a regular construct can be repeated. We refer by RE(#) to such regular expressions with numerical occurrence constraints.

We illustrate these additional operators in Figure 1. The formal definition is given in Section 2. Although the new operators can be expressed by the conventional regular operators, they cannot do so succinctly, which has severe implications on the complexity of the basic decision problems.

The goal of this paper is to study the complexity of the basic decision problems for DTDs, XSDs, and Relax NG with regular expressions extended with

interleaving and numerical occurrence constraints. The latter class of regular expressions is denoted by RE(#, &). As observed in Section 5, the complexity of inclusion and equivalence of RE(#, &)-expressions (and subclasses thereof) carries over to DTDs and single-type EDTDs. We therefore first establish the complexity of the basic decision problems for RE(#, &)-expressions and frequently occurring subclasses. These results are summarized in Table 1 and Table 2. Of independent interest, we introduce NFA(#, &)s, an extension of NFAs with counter and split/merge states for dealing with numerical occurrence constraints and interleaving operators. Finally, we revisit the simplification problem introduced in [28] for schemas with RE(#, &)-expressions. That is, given an extended DTD, can it be rewritten into an equivalent DTD or a single-type EDTD?

In this paper, we do not consider deterministic or one-unambiguous regular expressions which form a strict subclass of the regular expressions [7]. The reason is two-fold. First of all, one-unambiguity is a highly debatable constraint (cf., e.g., pg 98 of [40] and [24, 37]) which is only required for DTDs and XML Schema, not for Relax NG. Actually, the only direct advantage of one-unambiguity is that it gives rise to PTIME algorithms for some of the basic decision problems for standard regular expressions. The latter does not hold anymore for RE(#, &)-expressions rendering the notion even less attractive. Indeed, already intersection for one-unambiguous regular expressions is PSPACE-hard [27] and inclusion for one-unambiguous RE(#)-expressions is coNP-hard [18]. A second reason is that, in contrast to conventional regular expressions, one-unambiguity is not yet fully understood for regular expressions with numerical occurrence constraints and interleaving operators. Some initial results are provided by Bruggemann-Klein, and Kilpeläinen and Tuhkanen who give algorithms for deciding one-unambiguity of RE(&)- and RE(#)-expressions, respectively [5, 19]. No study investigating their properties has been undertaken. Such a study, although definitely relevant, is outside the scope of this paper.

Outline. In Section 2, we provide the necessary definitions. In Section 3, we define NFA(#, &). In Section 4 and Section 5, we establish the complexity of the basic decision problems for regular expressions and schema languages, respectively. We discuss simplification in Section 6. We conclude in Section 7. A version of this paper containing all proofs is available from the authors' webpages.

2 Definitions

2.1 Regular Expressions with Counting and Interleaving

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 usually denote the concatenation of w_1 and w_2 by $w_1 w_2$. The set of all strings is denoted by Σ^*. A

	INCLUSION	EQUIVALENCE	INTERSECTION
RE	PSPACE ([39])	PSPACE ([39])	PSPACE ([23])
RE(&)	EXPSPACE ([30])	EXPSPACE ([30])	**PSPACE**
RE(#) and RE(#, &)	**EXPSPACE**	**EXPSPACE**	**PSPACE**
NFA(#), NFA(&), and NFA(#, &)	**EXPSPACE**	**EXPSPACE**	**PSPACE**
DTDs with RE	PSPACE ([39])	PSPACE ([39])	PSPACE ([23])
DTDs with RE(#), RE(&), or RE(#, &)	**EXPSPACE**	**EXPSPACE**	**PSPACE**
single-type EDTDs with RE	PSPACE ([27])	PSPACE ([27])	EXPTIME ([27])
single-type EDTDs with RE(#), RE(&), or RE(#, &)	**EXPSPACE**	**EXPSPACE**	**EXPTIME**
EDTD with RE	EXPTIME ([36])	EXPTIME ([36])	EXPTIME ([35])
EDTDs with RE(#), RE(&), or RE(#, &)	**EXPSPACE**	**EXPSPACE**	**EXPTIME**

Table 1. Overview of new and known complexity results. All results are completeness results. The new results are printed in bold.

string language is a subset of Σ^*. For two string languages $L, L' \subseteq \Sigma^*$, we define their concatenation $L \cdot L'$ to be the set $\{w \cdot w' \mid w \in L, w' \in L'\}$. We abbreviate $L \cdot L \cdots L$ (i times) by L^i. By $w_1 \& w_2$ we denote the set of strings that is obtained by *interleaving* or *shuffling* w_1 and w_2 in every possible way. That is, for $w \in \Sigma^*$, $w \& \varepsilon = \varepsilon \& w = \{w\}$, and $a \cdot w_1 \& b \cdot w_2 = (\{a\} \cdot (w_1 \& b \cdot w_2)) \cup (\{b\} \cdot (a \cdot w_1 \& w_2))$. The operator & is then extended to languages in the canonical way.

The set of *regular expressions* over Σ, denoted by RE, is defined in the usual way: ε, and every Σ-symbol is a regular expression; and when r and s are regular expressions, then rs, $r + s$, and r^* are also regular expressions. By RE(#, &) we denote RE extended with two new operators: *interleaving* and *numerical occurrence constraints*. That is, when r and s are RE(#, &)-expressions then so are $r \,\&\, s$ and $r^{[k,\ell]}$ for $k, \ell \in \mathbb{N}$ with $k \leq \ell$ and $\ell > 0$. By RE(#) and RE(&), we denote RE extended only with counting and interleaving, respectively.

The language defined by a regular expression r, denoted by $L(r)$, is inductively defined as follows: $L(\varepsilon) = \{\varepsilon\}$; $L(a) = \{a\}$; $L(rs) = L(r) \cdot L(s)$; $L(r + s) = L(r) \cup L(s)$; $L(r^*) = \{\varepsilon\} \cup \bigcup_{i=1}^{\infty} L(r)^i$, $L(r^{[k,\ell]}) = \bigcup_{i=k}^{\ell} L(r)^i$; and, $L(r \,\&\, s) = L(r) \,\&\, L(s)$. The *size* of a regular expression r over Σ, denoted by $|r|$, is the number of Σ-symbols and operators occurring in r plus the sizes of the binary representations of the integers. By $r?$ and r^+, we abbreviate the expression $r + \varepsilon$ and rr^*, respectively. We assume familiarity with finite automata such as nondeterministic finite automata (NFAs) and deterministic finite automata (DFAs) [15].

2.2 Schema Languages for XML

The set of *unranked Σ-trees*, denoted by $\mathcal{T}_\Sigma$, is 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 $t_1, \ldots, t_n$ are attached to the root labeled a. We write a rather than $a()$. Notice that there is no a priori bound on the number of children of a node in a Σ-tree; such trees are therefore *unranked*. For every $t \in \mathcal{T}_\Sigma$, the *set of nodes* of t, denoted by $\mathrm{Dom}(t)$, is the set defined as follows: *(i)* if $t = \varepsilon$, then $\mathrm{Dom}(t) = \emptyset$; and *(ii)* if $t = a(t_1 \cdots t_n)$, where each $t_i \in \mathcal{T}_\Sigma$, then $\mathrm{Dom}(t) = \{\varepsilon\} \cup \bigcup_{i=1}^{n} \{iu \mid u \in \mathrm{Dom}(t_i)\}$. In the sequel, whenever we say tree, we always mean Σ-tree. A *tree language* is a set of trees.

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

Definition 1. Let $\mathcal{M}$ be a class of representations of regular string languages over Σ.

1. A *DTD($\mathcal{M}$)* 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 denote (Σ, d, s_d) by d and leave the start symbol s_d implicit whenever this cannot give rise to confusion.
   A tree t *satisfies* d if *(i)* $\mathrm{lab}^t(\varepsilon) = s_d$ 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.
2. An *extended DTD* (EDTD($\mathcal{M}$)) over Σ is a 5-tuple $D = (\Sigma, \Sigma', d, s, \mu)$, where Σ' is an alphabet of *types*, (Σ', d, s) is a DTD($\mathcal{M}$) over Σ', and μ is a mapping from Σ' to Σ.
   A tree t then *satisfies* an extended DTD if $t = \mu(t')$ for some $t' \in L(d)$. Here we abuse notation and let μ also denote its extension to define a homomorphism on trees. Again, we denote by $L(D)$ the set of trees satisfying D. For ease of exposition, we always take $\Sigma' = \{a^i \mid 1 \leq i \leq k_a, a \in \Sigma, i \in \mathbb{N}\}$ for some natural numbers k_a, and we set $\mu(a^i) = a$.
3. A *single-type EDTD* ($\mathrm{EDTD}^{\mathrm{st}}(\mathcal{M})$) over Σ is an EDTD($\mathcal{M}$) $D = (\Sigma, \Sigma', d, s, \mu)$ with the property that for every $a \in \Sigma'$, in the regular expression $d(a)$ no two types b^i and b^j with $i \neq j$ occur.

We denote by EDTD, EDTD(#), EDTD(&), and EDTD(#,&), the classes EDTD(RE), EDTD(RE(#)), EDTD(RE(&)), and EDTD(RE(#,&)), respectively. The same notation is used for $\mathrm{EDTD}^{\mathrm{st}}$ and DTDs.

For clarity, we write $a \to r$ rather than $d(a) = r$ in examples and proofs. Following this notation, a simple example of an EDTD is the following:

$$\begin{array}{l|l} \mathrm{shop}^1 \to (\mathrm{dvd}^1 + \mathrm{dvd}^2)^* \mathrm{dvd}^2 (\mathrm{dvd}^1 + \mathrm{dvd}^2)^* & \mathrm{title}^1 \to \varepsilon \\ \mathrm{dvd}^1 \to \mathrm{title}^1\ \mathrm{price}^1 & \mathrm{price}^1 \to \varepsilon \\ \mathrm{dvd}^2 \to \mathrm{title}^1\ \mathrm{price}^1\ \mathrm{discount}^1 & \mathrm{discount}^1 \to \varepsilon \end{array}$$

Here, dvd^1 defines ordinary DVDs, while dvd^2 defines DVDs on sale. The rule for shop^1 specifies that there should be at least one DVD on sale. Note that the above is not a single-type EDTD as dvd^1 and dvd^2 occur in the same rule.

As explained in [31, 28], EDTDs and single-type EDTDs correspond to Relax NG and XML Schema, respectively.

2.3 Decision Problems

The following problems are fundamental to this paper.

Definition 2. Let $\mathcal{M}$ be a class of regular expressions, string automata, or extended DTDs. We define the following problems:

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

We recall the known results concerning the complexity of REs and EDTDs.

Theorem 3. *(1)* INCLUSION, EQUIVALENCE, *and* INTERSECTION *for REs are* PSPACE*-complete [23, 39].*
(2) INCLUSION *and* EQUIVALENCE *for RE(&) are* EXPSPACE*-complete [30].*
(3) INCLUSION *and* EQUIVALENCE *for EDTDst are* PSPACE*-complete [27];* INTERSECTION *for EDTDst is* EXPTIME*-complete [27].*
(4) INCLUSION, EQUIVALENCE, *and* INTERSECTION *for EDTDs are* EXPTIME*-complete [35, 36].*
(5) MEMBERSHIP *for RE(&) is* NP*-complete [30].*

3 Automata for Occurrence Constraints and Interleaving

We introduce the automaton model NFA(#, &). In brief, an NFA(#, &) is an NFA with two additional features: *(i)* split and merge transitions to handle interleaving; and, *(ii)* counting states and transitions to deal with numerical occurrence constraints. The idea of split and merge transitions stems from Jędrzejowicz and Szepietowski [17]. Their automata are more general as they can express shuffle-closure which is not regular. Counting states are also used in the counter automata of Kilpeläinen and Tuhkanen [21], and Reuter [34] although these counter automata operate quite differently from NFA(#)s. Zilio and Lugiez [10] also proposed an automaton model that incorporates counting and interleaving by means of Presburger formulas. None of the cited papers consider the complexity of the basic decision problems of their model. We will use NFA(#, &)s for obtaining complexity upper bounds in Sections 4 and 5.

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

Definition 4. An NFA(#, &) is a 5-tuple $A = (Q, \Sigma, s, f, \delta)$ where

- Q is a finite set of states. To every $q \in Q$, we associate a lower bound $\min(q) \in \mathbb{N}$ and an upper bound $\max(q) \in \mathbb{N}$.
- $s, f \in Q$ is the start and final state, respectively.

– δ is the transition relation and is a subset of the union of the following sets:

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

Let $\max(A) = \max\{\max(q) \mid q \in Q\}$ be the largest upper bound occurring in A. A *configuration* γ is a pair (P, α) where, $P \subseteq Q$ is a set of states and $\alpha : Q \to \{0, \ldots, \max(A)\}$ is the value function mapping states to the value of their counter. For a state $q \in Q$, we denote by α_q the value function mapping q to 1 and every other state to 0. The initial configuration γ_s is $(\{s\}, \alpha_s)$. The final configuration γ_f is $(\{f\}, \alpha_f)$. When α is a value function then $\alpha[q = 0]$ and $\alpha[q^{++}]$ denote the functions obtained from α by setting the value of q to 0 and incrementing the value of q by 1, respectively, while leaving all other values unchanged.

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

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

1. **(ordinary transition)** there is a $q \in P$ and $(q, \sigma, q') \in \delta$ such that $\min(q) \leq \alpha(q) \leq \max(q)$, $P' = (P - \{q\}) \cup \{q'\}$, and $\alpha' = \alpha[q = 0][q'^{++}]$. That is, A is in state q and moves to state q' by reading σ (note that σ can be ε). The latter is only allowed when the counter value of q is between the lower and upper bound. The state q is replaced in P by q'. The counter of q is reset to zero and the counter of q' is incremented by one.
2. **(counting transition)** there is a $q \in P$ and $(q, \text{store}, q') \in \delta$ such that $\alpha(q) < \max(q)$, $P' = (P - \{q\}) \cup \{q'\}$, and $\alpha' = \alpha[q'^{++}]$. That is, A is in state q and moves to state q' by reading ε when the counter of q has not reached its maximal value yet. The state q is replaced in P by q'. The counter of q is not reset but remains the same. The counter of q' is incremented by one.
3. **(split transition)** there is a $q \in P$ and $(q, \text{split}, q'_1, q'_2) \in \delta$ such that $\min(q) \leq \alpha(q) \leq \max(q)$, $P' = (P - \{q\}) \cup \{q'_1, q'_2\}$, and $\alpha' = \alpha[q = 0][q'^{++}_1][q'^{++}_2]$. That is, A is in state q and splits into states q'_1 and q'_2 by reading ε when the counter value of q is between the lower and upper bound. The state q in P is replaced by (split into) q'_1 and q'_2. The counter of q is reset to zero, and the counters of q'_1 and q'_2 are incremented by one.

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

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

An NFA(#) is an NFA(#, &) without split and merge transitions. An NFA(&) is an NFA(#, &) without counting transitions. An NFA is an NFA(#) without counting transitions. NFA(#, &) therefore accept all regular languages.

The next theorem shows the complexity of translating between RE(#, &) and NFA(#, &), and NFA(#, &) and NFA. In brief, the proof of part (1) is by induction on the structure of RE(#, &)-expressions. Figure 2 illustrates the inductive steps for expressions $r_1^{[k,\ell]}$ and $r_1 \& r_2$, employing counter, and split and merge states, respectively. For part (2), we define an NFA from an NFA(#, &) that keeps in its state the current configuration of the latter: i.e., a set of states and a value function.

Theorem 5. *(1) Given an RE(#, &)-expression r, an equivalent NFA(#, &) can be constructed in time polynomial in the size of r.*
(2) Given an NFA(#, &) A, an equivalent NFA can be constructed in time exponential in the size of A.

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

Theorem 6. *(1)* EQUIVALENCE *and* INCLUSION *for NFA(#, &) is* EXPSPACE*-complete;*
(2) INTERSECTION *for NFA(#, &) is* PSPACE*-complete; and,*
(3) MEMBERSHIP *for NFA(#) is* NP*-hard,* MEMBERSHIP *for NFA(&), and NFA(#, &) is* PSPACE*-complete.*

We only provide some intuition. For part (1), membership in EXPSPACE follows directly from Theorem 5(2) and the fact that INCLUSION for NFAs is PSPACE-complete [39]. EXPSPACE-hardness follows from Theorem 5(1) and Theorem 7(3). For part (2), PSPACE-hardness follows from PSPACE-hardness of INTERSECTION for REs [23]. Membership in PSPACE is witnessed by an in parallel simulation of the given NFA(#, &)s on a guessed string. Finally, NP-hardness of MEMBERSHIP for NFA(#)s is by a reduction from INTEGER KNAPSACK, PSPACE-hardness of MEMBERSHIP for NFA(&)s is by a reduction from CORRIDOR TILING.

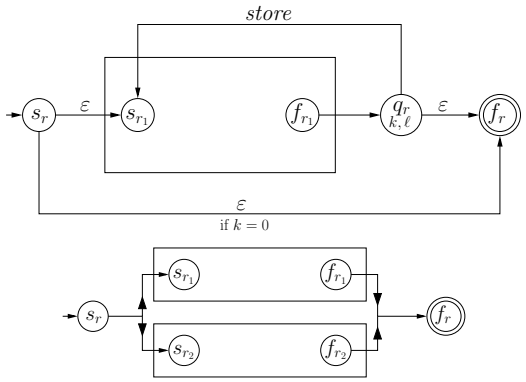

Fig. 2. From RE(#, &) to NFA(#, &).

4 Complexity of Regular Expressions

Before we turn to schemas, we first deal with the complexity of regular expressions and frequently used subclasses.

Mayer and Stockmeyer already established the EXPSPACE-completeness of INCLUSION and EQUIVALENCE for RE(&) [30]. From Theorem 5(1) and Theorem 6(1) it then directly follows that adding numerical occurrence constraints does not increase the complexity. It further follows from Theorem 5(1) and Theorem 6(2), that INTERSECTION for RE(#, &) is in PSPACE. We stress that the latter results could also have been obtained without making use of NFA(#, &) but by translating RE(#, &)s directly to NFAs. However, in the case of INTERSECTION such a construction should be done in an on-the-fly fashion in order not to go beyond PSPACE. Although such an approach is possible, we prefer the shorter and more elegant construction using NFA(#, &)s. Finally, we show that INCLUSION and EQUIVALENCE of RE(#) is also EXPSPACE-hard. While Mayer and Stockmeyer reduce from REs with intersection [12], we employ a reduction from EXP-CORRIDOR TILING.

Theorem 7. *1.* EQUIVALENCE *and* INCLUSION *for RE(#, &) is in* EXPSPACE*;*
2. INTERSECTION *for RE(#, &) is* PSPACE*-complete; and,*
3. EQUIVALENCE *and* INCLUSION *for RE(#) is* EXPSPACE*-hard.*

Proof. We prove (3). It suffices to show that it is EXPSPACE-hard to decide whether a given RE(#) defines Σ^*. The proof is a reduction from EXP-CORRIDOR TILING. A *tiling instance* is a tuple $T = (X, H, V, x_\bot, x_\top, n)$ where X is a finite set of tiles, $H, V \subseteq X \times X$ are the horizontal and vertical constraints, $x_\bot, x_\top \in X$, and n is a natural number in unary notation. A *correct exponential corridor tiling for* T is a mapping $\lambda : \{1, \ldots, m\} \times \{1, \ldots, 2^n\} \to X$ for some $m \in \mathbb{N}$ such that the following constraints are satisfied:

– the first tile of the first row is $x_\bot$: $\lambda(1, 1) = x_\bot$;

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

The EXP-CORRIDOR TILING problem asks, given a tiling instance, whether there exists a correct exponential corridor tiling. The latter problem is easily shown to be EXPSPACE-complete [41].

We proceed with the reduction from EXP-CORRIDOR TILING. Thereto, let $T = (X, H, V, x_\bot, x_\top, n)$ be a tiling instance. We construct an RE(#)-expression r which defines the set of all strings iff there is no correct tiling for T. As EXPSPACE is closed under complement, the EXPSPACE-hardness of EQUIVALENCE and INCLUSION for RE(#) follows.

Let $\Sigma = X \cup \{\triangle\}$. For a set $S = \{s_1, \ldots, s_k\} \subseteq \Sigma$, we abuse notation and abbreviate $(s_1 + \cdots + s_k)$ simply by S. We represent a candidate tiling consisting of m rows $\rho_1, \ldots, \rho_m$ by the string $\triangle\rho_1\triangle\cdots\triangle\rho_m\triangle$. Here, every two successive rows are delimited by the symbol $\triangle$. We now define r as a disjunction of RE(#)-expressions where every disjunct catches an error in the candidate tiling. Therefore, when r is equivalent to Σ^* there can be no correct tiling for T. It remains to define the disjuncts constituting r:

1. The string does not start or end with $\triangle$: $X\Sigma^* + \Sigma^* X$.
2. There are no 2^n tiles between two successive delimiters: $\Sigma^*\triangle(X^{[0,2^n-1]} + X^{[2^n+1,2^n+1]}X^*)\triangle\Sigma^*$.
3. The first tile is not $x_\bot$: $\triangle x\Sigma^*$ for every $x \neq x_\bot$.
4. The first tile of the last row is not $x_\top$: $\Sigma^*\triangle x X^*\triangle$ for every $x \neq x_\top$.
5. Horizontal constraint violation: $\Sigma^* x_1 x_2 \Sigma^*$ for every $(x_1, x_2) \notin H$.
6. Vertical constraint violation: $\Sigma^* x_1 \Sigma^{[2^n,2^n]} x_2 \Sigma^*$ for every $(x_1, x_2) \notin V$.

Clearly, a Σ-string that does not satisfy any of the disjuncts in r is a correct tiling for T. Hence, $L(r) \neq \Sigma^*$ iff there is a correct tiling for T. □

Bex et al. [4] established that the far majority of regular expressions occurring in practical DTDs and XSDs are of a very restricted form as defined next. The class of *chain regular expressions (CHAREs)* are those REs consisting of a sequence of factors $f_1 \cdots f_n$ where every factor is an expression of the form $(a_1{+}\cdots{+}a_n)$, $(a_1{+}\cdots{+}a_n)?$, $(a_1{+}\cdots{+}a_n)^+$, or, $(a_1{+}\cdots{+}a_n)^*$, where $n \geq 1$ and every a_i is an alphabet symbol. For instance, the expression $a(b+c)^*d^+(e+f)?$ is a CHARE, while $(ab+c)^*$ and $(a^*+b?)^*$ are not.[1]

We introduce some additional notation to define subclasses and extensions of CHAREs. By CHARE(#) we denote the class of CHAREs where also factors of the form $(a_1 + \cdots + a_n)^{[k,\ell]}$ are allowed. For the following fragments, we list the admissible types of factors. Here, a, $a?$, a^* denote the factors $(a_1 + \cdots + a_n)$, $(a_1 + \cdots + a_n)?$, and $(a_1 + \cdots + a_n)^+$, respectively, with $n = 1$, while $a\#$ denotes $a^{[k,\ell]}$, and $a\#^{>0}$ denotes $a^{[k,\ell]}$ with $k > 0$.

[1] We disregard here the additional restriction used in [3] that every symbol can occur only once.

	INCLUSION	EQUIVALENCE	INTERSECTION
CHARE	PSPACE [27]	in PSPACE [39]	PSPACE [27]
CHARE(#)	**EXPSPACE**	**in EXPSPACE**	**PSPACE**
CHARE($a, a?$)	CONP [27]	in PTIME [27]	NP [27]
CHARE(a, a^*)	CONP [27]	in PTIME [27]	NP [27]
CHARE($a, a?, a\#$)	**PSPACE-hard / in EXPSPACE**	**in PTIME**	**NP**
CHARE($a, a\#^{>0}$)	**in PTIME**	**in PTIME**	**in PTIME**

Table 2. Overview of new and known complexity results concerning Chain Regular Expressions. All results are completeness results, unless otherwise mentioned. The new results are printed in bold.

Table 2 lists the new and the relevant known results. We first show that adding numerical occurrence constraints to CHAREs increases the complexity of INCLUSION by one exponential. Again we reduce from EXP-CORRIDOR TILING.

Theorem 8. INCLUSION *for CHARE(#) is* EXPSPACE*-complete.*

Adding numerical occurrence constraints to the fragment CHARE($a, a?$) and CHARE(a, a^*), makes INCLUSION PSPACE-hard but keeps EQUIVALENCE in PTIME and INTERSECTION in NP.

Theorem 9. *(1)* EQUIVALENCE *for CHARE($a, a?, a\#$) is in* PTIME*.*
(2) INCLUSION *for CHARE($a, a?, a\#$) is* PSPACE*-hard and in* EXPSPACE*.*
(3) INTERSECTION *for CHARE($a, a?, a\#$) is* NP*-complete.*

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

Theorem 10. INCLUSION*,* EQUIVALENCE*, and* INTERSECTION *for CHARE($a, a\#^{>0}$) are in* PTIME*.*

5 Complexity of Schemas

5.1 DTDs and Single-Type EDTDs

In [27] it was shown for any subclass of the REs that the complexity of INCLUSION and EQUIVALENCE is the same as the complexity of the corresponding problem for DTDs and single-type EDTDs. We next generalize this result to RE(#, &). As a corollary, all results of the previous section carry over to DTDs and single-type DTDs. The same holds for INTERSECTION and DTDs.

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 [14]. For our purposes, it is sufficient that important complexity classes like PTIME, NP, CONP, PSPACE, and EXPSPACE have this property, and that every such class contains PTIME.

Proposition 11. *Let $\mathcal{R}$ be a subclass of RE(#, &) and let $\mathcal{C}$ be a complexity class closed under positive reductions. Then the following are equivalent:*

(a) INCLUSION *for $\mathcal{R}$ expressions is in $\mathcal{C}$.*
(b) INCLUSION *for DTD($\mathcal{R}$) is in $\mathcal{C}$.*
(c) INCLUSION *for EDTDst($\mathcal{R}$) is in $\mathcal{C}$.*

The corresponding statement holds for EQUIVALENCE.

The previous proposition can be generalized to INTERSECTION of DTDs as well. The proof carries over literally from [27].

Proposition 12. *Let $\mathcal{R}$ be a subclass of RE(#, &) and let $\mathcal{C}$ be a complexity class which is closed under positive reductions. Then the following are equivalent:*

(a) INTERSECTION *for $\mathcal{R}$ expressions is in $\mathcal{C}$.*
(b) INTERSECTION *for DTD($\mathcal{R}$) is in $\mathcal{C}$.*

The above proposition does not hold for single-type EDTDs. Indeed, there is a class of regular expressions $\mathcal{R}'$ for which INTERSECTION is NP-complete while INTERSECTION for EDTD$^{\text{st}}(\mathcal{R}')$ is EXPTIME-complete [27].

5.2 Extended EDTDs

We next consider the complexity of the basic decision problems for EDTDs with numerical occurrence constraints and interleaving. As the basic decision problems are EXPTIME-complete for EDTD(RE), the straightforward approach of translating every RE(#, &)-expression into an NFA and then applying the standard algorithms gives rise to a double exponential time complexity. By using NFA(#, &), we can do better: EXPSPACE for INCLUSION and EQUIVALENCE, and, more surprisingly, EXPTIME for INTERSECTION.

Theorem 13. *(1)* EQUIVALENCE *and* INCLUSION *for EDTD(#,&) is in* EXPSPACE*;*
(2) EQUIVALENCE *and* INCLUSION *for EDTD(#) and EDTD(&) is* EXPSPACE*-hard; and,*
(3) INTERSECTION *for EDTD(#,&) is* EXPTIME*-complete.*

Proof (Sketch).
(1) Given two EDTDs $D_1 = (\Sigma, \Sigma_1', d_1, s_1, \mu_1)$ and $D_2 = (\Sigma, \Sigma_2', d_2, s_2, \mu_2)$, we compute a set E of pairs $(C_1, C_2) \in 2^{\Sigma_1'} \times 2^{\Sigma_2'}$ where $(C_1, C_2) \in E$ iff there exists a tree t such that $C_j = \{\tau \in \Sigma_j' \mid t \in L((D_j, \tau))\}$ for each $j = 1, 2$. Here, (D_j, τ) denotes the EDTD D_j with start symbol τ. So, every C_j is the set of types that can be assigned by D_j to the root of t. Or when viewing D_j as a tree automaton, C_j is the set of states that can be assigned to the root in a run on t. The tree t is called a *witness tree*. Then, $t \in L(D_1)$ (resp., $t \in L(D_2)$) if $s_1 \in C_1$ (resp. $s_2 \in C_2$). Hence, $L(D_1) \not\subseteq L(D_2)$ iff there exists a pair $(C_1, C_2) \in E$ with $s_1 \in C_1$ and $s_2 \notin C_2$.

Although each witness tree can have exponential depth and therefore double exponential size, we do not need to compute it directly. Instead, we compute the set E in a bottom-up fashion where we make use of an NFA(#, &)-representation of the RE(#, &)-expressions.

(2) Is immediate from Theorem 3(2) and Theorem 7(2).

(3) In brief, given a set of EDTDs, we construct an alternating polynomial space TM which incrementally guesses a tree defined by all schemas. To be precise, the algorithm guesses the first-child-next-sibling encoding of the unranked tree. Again, RE(#, &)-expressions are translated into equivalent NFA(#, &)s.□

6 Simplification

The simplification problem is defined as follows: Given an EDTD, check whether it has an equivalent EDTD of a restricted type, i.e., an equivalent DTD or single-type EDTD. In [28], this problem was shown to be EXPTIME-complete for EDTDs with standard regular expressions. We revisit this problem in the context of RE(#, &).

Theorem 14. Given an EDTD(#,&), deciding whether it is equivalent to an EDTD$^{\text{st}}$(#,&) or DTD(#,&) is EXPSPACE-complete.

Proof (Sketch). We only show that the problem is hard for EXPSPACE. We use a reduction from universality of RE(#, &), i.e., deciding whether an RE(#, &)-expression is equivalent to Σ^*. The proof of Theorem 7(2) shows that the latter is EXPSPACE-hard. To this end, let r be an RE(#, &)-expression over Σ and let b and s be two symbols not occurring in Σ. By definition, $L(r) \neq \emptyset$. Define $D = (\Sigma \cup \{b, s\}, \Sigma \cup \{s, b^1, b^2\}, d, s, \mu)$ as the EDTD with the following rules: $s \to (b^1)^* b^2 (b^1)^*$, $b^1 \to \Sigma^*$, and $b^2 \to r$, where for every $\tau \in \Sigma \cup \{s\}$, $\mu(\tau) = \tau$, and $\mu(b^1) = \mu(b^2) = b$. We claim that D is equivalent to a single-type DTD or a DTD iff $L(r) = \Sigma^*$. Clearly, if r is equivalent to Σ^*, then D is equivalent to the DTD (and therefore also to a single-type EDTD) with rules: $s \to b^*$ and $b \to \Sigma^*$. Conversely, suppose that there exists an EDTD$^{\text{st}}$ which defines the language $L(D)$. Towards a contradiction, assume that r is not equivalent to Σ^*. Let w_r be a string in $L(r)$ and let $w_{\neg r}$ be a Σ-string not in $L(r)$. Consider the trees $t_1 = s(b(w_r)b(w_{\neg r}))$ and $t_2 = s(b(w_{\neg r})b(w_r))$. Clearly, t_1 and t_2 are in $L(D)$. However, the tree $t = s(b(w_{\neg r})b(w_{\neg r}))$ obtained from t_1 by replacing its left subtree by the left subtree of t_2 is not in $L(D)$. According to Theorem 7.1 in [28], every tree language defined by a single-type EDTD is closed under such an exchange of subtrees. So, this means that $L(D)$ cannot be defined by an EDTD$^{\text{st}}$, which leads to the desired contradiction. □

7 Conclusion

The present work gives an overview of the complexity of the basic decision problems for abstractions of several schema languages including numerical occurrence

constraints and interleaving. W.r.t. INTERSECTION the complexity remains the same, while for INCLUSION and EQUIVALENCE the complexity increases by one exponential for DTDs and single-type EDTDs, and goes from EXPTIME to EXPSPACE for EDTDs. The results w.r.t. CHAREs also follow this pattern. We further showed that the complexity of simplification increases to EXPSPACE.

We emphasize that this is a theoretical study delineating the worst case complexity boundaries for the basic decision problems. Although these complexities must be studied, we note that the regular expressions used in the hardness proofs do not correspond at all to those employed in practice. Further, w.r.t. XSDs, our abstraction is not fully adequate as we do not consider the one-unambiguity (or unique particle attribution) constraint. However, it is doubtful that this constraint is the right one to get tractable complexities for the basic decision problems. Indeed, already intersection for unambiguous regular expressions is PSPACE-hard [27] and inclusion for one-unambiguous RE(#)-expressions is coNP-hard [18]. It would therefore be desirable to find robust subclasses for which the basic decision problems are in PTIME.

References

1. S. Abiteboul, P. Buneman, and D. Suciu. *Data on the Web : From Relations to Semistructured Data and XML*. Morgan Kaufmann, 1999.
2. M. Benedikt, W. Fan, and F. Geerts. XPath satisfiability in the presence of DTDs. In *PODS 2005*, pages 25–36, 2005.
3. G.J. Bex, F. Neven, T. Schwentick, and K. Tuyls. Inference of concise DTDs from XML data. In *VLDB 2006*, pages 115–126, 2006.
4. G.J. Bex, F. Neven, and J. Van den Bussche. DTDs versus XML schema: A practical study. In *WebDB 2004*, pages 79–84, 2004.
5. A. Brüggemann-Klein. Unambiguity of extended regular expressions in SGML document grammars. In *ESA 1993*, pages 73–84, 1993.
6. A. Brüggemann-Klein, M. Murata, and D. Wood. Regular tree and regular hedge languages over unranked alphabets: Version 1, april 3, 2001. Technical Report HKUST-TCSC-2001-0, The Hongkong University of Science and Technology, 2001.
7. A. Brüggemann-Klein and D. Wood. One-unambiguous regular languages. *Information and Computation*, 142(2):182–206, 1998.
8. J. Clark and M. Murata. *RELAX NG Specification*. OASIS, December 2001.
9. J. Cristau, C. Löding, and W. Thomas. Deterministic automata on unranked trees. In *FCT 2005*, pages 68–79. Springer, 2005.
10. S. Dal-Zilio and D. Lugiez. XML schema, tree logic and sheaves automata. In *RTA*, pages 246–263, 2003.
11. Alin Deutsch, Mary F. Fernandez, and Dan Suciu. Storing Semistructured Data with STORED. In *SIGMOD 1999*, pages 431–442, 1999.
12. M. Fürer. The complexity of the inequivalence problem for regular expressions with intersection. In *ICALP 1980*, pages 234–245. Springer, 1980.
13. M.R. Garey and D.S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. Freeman, 1979.
14. L. Hemaspaandra and M. Ogihara. *Complexity Theory Companion*. Springer, 2002.
15. J.E. Hopcroft, R. Motwani, and J.D. Ullman and. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, second edition, 2001.

16. H. Hosoya and B. C. Pierce. XDuce: A statically typed XML processing language. *ACM Trans. Inter. Tech.*, 3(2):117–148, 2003.
17. J. Jędrzejowicz and A. Szepietowski. Shuffle languages are in P. *Theoretical Computer Science*, 250(1-2):31–53, 2001.
18. P. Kilpeläinen. Inclusion of unambiguous #REs is NP-hard. Unpublished note, University of Kuopio, Finland, May 2004.
19. P. Kilpeläinen and R. Tuhkanen. One-unambiguity of regular expressions with numeric occurrence indicators. Tech. Rep. A/2006/2, Univ. Kuopio, Finland, 2006.
20. P. Kilpeläinen and R. Tuhkanen. Regular expressions with numerical occurrence indicators — preliminary results. In *SPLST 2003*, pages 163–173, 2003.
21. P. Kilpeläinen and R. Tuhkanen. Towards efficient implementation of XML schema content models. In *DOCENG 2004*, pages 239–241. ACM, 2004.
22. C. Koch, S. Scherzinger, N. Schweikardt, and B. Stegmaier. Schema-based scheduling of event processors and buffer minimization for queries on structured data streams. In *VLDB 2004*, pages 228–239, 2004.
23. D. Kozen. Lower bounds for natural proof systems. In *FOCS 1977*, pages 254–266. IEEE, 1977.
24. M. Mani. Keeping chess alive — Do we need 1-unambiguous content models? In *Extreme Markup Languages*, Montreal, Canada, 2001.
25. I. Manolescu, D. Florescu, and D. Kossmann. Answering XML Queries on Heterogeneous Data Sources. In *VLDB 2001*, pages 241–250, 2001.
26. W. Martens and F. Neven. Frontiers of tractability for typechecking simple XML transformations. *Journal of Computer and System Sciences*, 2006. To Appear.
27. W. Martens, F. Neven, and T. Schwentick. Complexity of decision problems for simple regular expressions. In *MFCS 2004*, pages 889–900, Berlin, 2004. Springer.
28. W. Martens, F. Neven, T. Schwentick, and G. J. Bex. Expressiveness and complexity of XML schema. *ACM Trans. Database Systems*, 31(3), 2006. To appear.
29. W. Martens and J. Niehren. Minimizing tree automata for unranked trees. In *DBPL 2005*, pages 232–246, 2005.
30. A. J. Mayer and L. J. Stockmeyer. Word problems — this time with interleaving. *Information and Computation*, 115(2):293–311, 1994.
31. M. Murata, D. Lee, M. Mani, and K. Kawaguchi. Taxonomy of XML schema languages using formal language theory. *ACM Trans. Inter. Tech.*, 5(4):1–45, 2005.
32. F. Neven and T. Schwentick. XPath containment in the presence of disjunction, DTDs, and variables. *Logical Methods in Computer Science*, page To appear, 2006.
33. Y. Papakonstantinou and V. Vianu. DTD inference for views of XML data. In *PODS 2000*, pages 35–46, New York, 2000. ACM Press.
34. F. Reuter. An enhanced W3C XML Schema-based language binding for object oriented programming languages. Manuscript, 2006.
35. H. Seidl. Deciding equivalence of finite tree automata. *SIAM Journal on Computing*, 19(3):424–437, 1990.
36. H. Seidl. Haskell overloading is DEXPTIME-complete. *Information Processing Letters*, 52(2):57–60, 1994.
37. C.M. Sperberg-McQueen. XML Schema 1.0: A language for document grammars. In *XML 2003*, 2003.
38. C.M. Sperberg-McQueen and H. Thompson. XML Schema. http://www.w3.org/XML/Schema, 2005.
39. L.J. Stockmeyer and A.R. Meyer. Word problems requiring exponential time: Preliminary report. In *STOC 1973*, pages 1–9. ACM Press, 1973.
40. E. van der Vlist. *XML Schema*. O'Reilly, 2002.

41. P. van Emde Boas. The convenience of tilings. In *Complexity, Logic and Recursion Theory*, volume 187 of *Lec. Notes in Pure and App. Math.*, pages 331–363. 1997.
42. G. Wang, M. Liu, J. X. Yu, B. Sun, G. Yu, J. Lv, and H. Lu. Effective schema-based XML query optimization techniques. In *IDEAS 2003*, pages 230–235, 2003.

Appendix

Proofs for Section 3

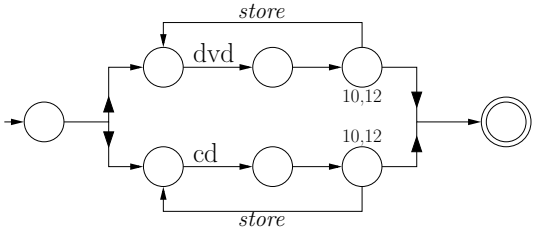

Fig. 3. An NFA$(\#, \&)$ for the language $\text{dvd}^{[10,12]}$ & $\text{cd}^{[10,12]}$. For readability, we only displayed the alphabet symbol on non-epsilon transitions and counters for states q where $\min(q)$ or $\max(q)$ are different from one. The arrows from the initial state and to the final state are split and merge transitions, respectively. The arrows labeled *store* represent counting transitions.

Proof of Theorem 5:

(1) Given an RE$(\#, \&)$-expression r, an equivalent NFA$(\#, \&)$ can be constructed in time polynomial in the size of r.
(2) Given an NFA$(\#, \&)$ A, an equivalent NFA can be constructed in time exponential in the size of A.

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

For r of the form ε, a, $r_1 \cdot r_2$, $r_1 + r_2$ and r_1^* these are the usual RE to NFA with ε-transition constructions as displayed in text books such as [15].

We perform the following steps for the numerical occurrence and interleaving operator which are graphically illustrated in Figure 2.

(i) If $r = (r_1)^{[k,\ell]}$ and $A(r_1) = (Q_1, \Sigma, s_1, f_1, \delta_1)$, then
- $Q_r := Q_{r_1} \uplus \{s_r, f_r, q_r\}$;
- $\min(s_r) = \max(s_r) = \min(f_r) = \max(f_r) = 1$, $\min(q_r) = k$, and $\max(q_r) = \ell$;
- if $k \neq 0$ then $\delta_r := \delta_{r_1} \uplus \{(s_r, \varepsilon, s_{r_1}), (f_{r_1}, \varepsilon, q_r), (q_r, \text{store}, s_{r_1}), (q_r, \varepsilon, f_r)\}$; and,

– if $k = 0$ then $\delta_r := \delta_{r_1} \uplus \{(s_r, \varepsilon, s_{r_1}), (f_{r_1}, \varepsilon, q_r), (q_r, \text{store}, s_{r_1}), (q_r, \varepsilon, f_r), (s_r, \varepsilon, f_r)\}$.

(ii) If $r = r_1 \,\&\, r_2$, $A(r_1) = (Q_{r_1}, \Sigma, s_{r_1}, f_{r_1}, \delta_{r_1})$ and $A(r_2) = (Q_{r_2}, \Sigma, s_{r_2}, f_{r_2}, \delta_{r_2})$, then
– $Q_r := Q_{r_1} \uplus Q_{r_2} \uplus \{s_r, f_r\}$;
– $\min(s_r) = \max(s_r) = \min(f_r) = \max(f_r) = 1$;
– $\delta_r := \delta_{r_1} \uplus \delta_{r_2} \uplus \{(s_r, \text{split}, s_{r_1}, s_{r_2}), (f_{r_1}, f_{r_2}, \text{merge}, f_r)\}$.

Notice that in each step of the construction, a constant number of states are added to the automaton. Moreover, the constructed counters are linear in the size of r. It follows that the size of $A(r)$ is linear in the size of r.

We argue that the construction is correct by induction on the structure of r. When $r = a$ or $r = \varepsilon$, the correctness is immediate from the standard construction. We proceed with the induction step for the numerical and interleaving operators. In both cases, correctness can be shown by observing the sequences of transitions that take automata from their initial to their final configuration.

(i) Let $r = (r_1)^{[k,\ell]}$ and w be a string. We show that $w \in L(r)$ iff $w \in L(A(r))$. If $k = 0$ and $w = \varepsilon$, we have that $w \in L(r)$ iff $w \in L(A(r))$ by construction, since (s_r, ε, f_r) is a transition in $A(r)$. If $k \neq 0$ or $w \neq \varepsilon$, then $w \in L(r)$ iff there exists an $n \in \{k, \ldots, \ell\}$ and strings $w_1, \ldots, w_n \in L(r_1)$ such that $w = w_1 \cdots w_n$. By induction, for each $i = 1, \ldots, n$, $w_i \in L(r_1)$ iff $w_i \in L(A(r_1))$. For each $i = 1, \ldots, n$, $w_i \in L(A(r_1))$ iff there exists a sequence T_i of transitions of $A(r_1)$ that takes $A(r_1)$ from its initial to its final configuration while reading w_i. The latter sequences of transitions exist for each $i = 1, \ldots, n$ iff there exists a sequence of transitions $T = (s, \varepsilon, s_{r_1})T_1(f_{r_1}, \varepsilon, q_r)(q_r, \text{store}, s_{r_1})T_2 \cdots T_n(f_{r_1}, \varepsilon, q_r)(q_r, \varepsilon, f)$ that takes $A(r)$ from its initial to its final configuration while reading w.

(ii) Let $r = r_1 \,\&\, r_2$ and let w be a string. We show that $w \in L(r)$ iff $w \in L(A(r))$. We have that $w \in L(r)$ iff there exist $w_1 \in L(r_1)$ and $w_2 \in L(r_2)$ such that $w \in w_1 \,\&\, w_2$. By induction, $w_1 \in L(r_1)$ and $w_2 \in L(r_2)$ iff $w_1 \in L(A(r_1))$ and $w_2 \in L(A(r_2))$. For each $j = 1, 2$, $w_j \in L(A(r_j))$ iff there exists a sequence T_j of transitions that take $A(r_j)$ from its initial to its final configuration while reading w_j. The latter is the case if and only if there exists a sequence of transitions $(s, \text{split}, s_{r_1}, s_{r_2})T(f_{r_1}, f_{r_2}, \text{merge}, f)$ that takes $A(r)$ from s to f while reading r, such that the concatenation of the labels in T is w.

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

– $Q_B = 2^Q \times (\{1, \ldots, \max(A)\}^{Q_A})$;
– $s_B = (\{s_A\}, \alpha_{s_A})$;
– $f_B = (\{f_A\}, \alpha_{f_A})$;
– $\delta_B = \{\big((P_1, \alpha_1), \sigma, (P_2, \alpha_2)\big) \mid \sigma \in \Sigma_\varepsilon$ and $(P_1, \alpha_1) \rightarrow_{A,\sigma} (P_2, \alpha_2)$ for configurations (P_1, α_1) and (P_2, α_2) of $A\}$.

Obviously, B can be constructed from A in exponential time. Notice that the size of Q_B is smaller than $2^{|Q_A|}.2^{|A|\cdot|Q_A|}$. Furthermore, as the transition relation of B is isomorphic to the union of the relations $\rightarrow_{A,\sigma}$ over all $\sigma \in \Sigma_\varepsilon$, it is immediate that $L(A) = L(B)$. □

Before giving the proof of Theorem 6, we describe some new merge and split transitions which can be written in function of the regular split and merge transitions. These transitions will be used in the proof of Theorem-6(3).

1. $(q_1, q_2, \text{merge-split}, q'_1, q'_2)$: States q_1 and q_2 are read, and immediately split into states q'_1 and q'_2.
2. $(q_1, q_2, q_3, \text{merge-split}, q'_1, q'_2, q'_3)$: States q_1, q_2 and q_3 are read, and immediately split into states q'_1, q'_2 and q'_3.
3. $(q_1, \text{split}, q'_1, \ldots, q'_n)$: State q_1 is read, and is immediately split into states $q'_1, \ldots, q'_n$.
4. $(q_1, \ldots, q_n, \text{merge}, q'_1)$: States $q_1, \ldots, q_n$ are read, and are merged into state q'_1.

Transitions of type 1 (resp. 2) can be rewritten using 2 (resp. 4) *regular* transitions, and 1 (resp. 3) new helping states. Transitions of type 3 and 4 can be rewritten using $(n-1)$ regular transitions and $(n-1)$ new helping states. For example, the transition $(q_1, q_2, \text{merge-split}, q'_1, q'_2)$ is equal to the transitions $(q_1, q_2, \text{merge}, q_h)$, and $(q_h, \text{split}, q'_1, q'_2)$, where q_h is a new helping state.

Proof of Theorem 6:

(1) EQUIVALENCE *and* INCLUSION *for NFA*$(\#, \&)$ *is* EXPSPACE*-complete;*
(2) INTERSECTION *for NFA*$(\#, \&)$ *is* PSPACE*-complete; and,*
(3) MEMBERSHIP *for NFA*$(\#)$ *is* NP*-hard and* MEMBERSHIP *for NFA*$(\&)$ *and NFA*$(\#, \&)$ *is* PSPACE*-complete.*

Proof. (1) EXPSPACE-hardness follows from Theorem 5(1) and Theorem 7(3). Membership in EXPSPACE follows from Theorem 5(2) and the fact that INCLUSION for NFAs is PSPACE-complete [39].

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

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

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

(3) The MEMBERSHIP problem for NFA(#, &)s is easily seen to be in PSPACE by an on-the-fly implementation of the construction in Theorem 5(2). Indeed, as a configuration of an NFA(#, &) $A = (Q, \Sigma, s, f, \delta)$ has size as most $|Q| + |Q| \cdot \max(A)$, we can store a configuration using only polynomial space.

We show that the MEMBERSHIP problem for NFA(#)s is NP-hard by a reduction from a modification of INTEGER KNAPSACK. We define this problem as follows. Given a set of natural numbers $W = \{w_1, \ldots, w_k\}$ and two integers m and n, the problem asks whether there exists a mapping $\tau : W \to \mathbb{N}$ such that $m \leq \sum_{w \in W} \tau(w) \times w \leq n$. The latter mapping is called a solution. This problem is known to be NP-complete [13].

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

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

Formally, the transitions of A are the following:

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

We set $\min(q) = m$, $max(q) = n$ and $\min(q_{w_i}) = \max(q_{w_i}) = w_i$ for each q_{w_i}.

Finally, we show that MEMBERSHIP for NFA(&)s is PSPACE-hard. The proof is a reduction from CORRIDOR TILING. A *tiling instance* is a tuple $T = (X, H, V, \overline{b}, \overline{t}, n)$ where X is a finite set of tiles, $H, V \subseteq X \times X$ are the horizontal and vertical constraints, and $\overline{b}, \overline{t}$ are n-tuples of tiles ($\overline{b}$ and $\overline{t}$ stand for *bottom row* and *top row*, respectively).

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

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

The CORRIDOR TILING problem asks, given a tiling instance, whether there exists a correct corridor tiling. The latter problem is PSPACE-complete [41].

Given a tiling instance T, we will construct an NFA(&)A over the empty alphabet ($\Sigma = \emptyset$), which accepts ε iff there exists a correct corridor tiling for T. Our automaton will try to construct a correct tiling for T, and accept when it finds such a tiling.

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

The only question left is how A can transform its state set which describes the current row, into a state set which describes a valid row on top of the current row. We do this one tile at a time, and begin with the first tile, say x_i, in the current row. This tile is represented by x_i^1 in the state set. Now, for every tile x_j, for which $(x_i, x_j) \in V$, we allow x_i^1 to be replaced by x_j^1, since x_j can be the first tile of the row on top of the current row. For the second tile of the next row, we have to replace the second tile of the current row, say x_k, by a new tile, say x_l, such that the vertical constraints between x_k and x_l are satisfied and such that the horizontal constraints between x_l and the tile we just placed at the first position of the first row, x_j, are satisfied.

In this manner, we run through the whole row and replace it by a new one. To do this properly, we need to know at any time at which position we have to update the tile. Therefore, we create an extra set of states $p_1, \ldots, p_n$, where the state p_i says that the tile at position i has to be updated. So, the state set always consists of a state p_i, and a number of states which represent the current and next row. Here, the states up to position i already represent the tiles of the next row, the states from position i still represent the current row, and i is the next position where we have to update the tile.

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

- $Q = \{x^j | x \in X, 1 \leq j \leq n\} \cup \{p_i | 1 \leq i \leq n\} \cup \{s, f\}$
- $\Sigma = \emptyset$
- δ is the union of the following
  - $(s, \text{split}, p_1, \overline{b}_1^1, \ldots, \overline{b}_n^n)$: From the initial state we immediately go to the states which represent the bottom row.
  - $(p_1, \overline{t}_1^1, \ldots, \overline{t}_n^n, \text{merge}, f)$: When the state set represents a full row (we are in state p_1), and the current row is the accepting row, we merge all the states to the accepting state.
  - $\forall x_i, x_j \in X, (x_j, x_i) \in V$: $(p_1, x_j^1, \text{merge-split}, p_2, x_i^1)$: When we have to update the first tile, we only have to check the vertical constraints with the first tile of the previous row.
  - $\forall x_i, x_j, x_k \in X, m \in \mathbb{N}, 2 \leq m \leq n, (x_k, x_i) \in V, (x_j, x_i) \in H$: $(p_m, x_k^m, x_j^{m-1}, \text{merge-split}, p_{(m \mod n)+1}, x_i^m, x_j^{m-1})$: When we have to

update a tile at the nth ($n \neq 1$) position, we have to check the vertical constraint with the nth tile at the previous row, and the horizontal constraint with the $(n-1)$th tile of the new row.

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

Proofs for Section 4

Proof of Theorem 7:

(1) EQUIVALENCE *and* INCLUSION *for RE(#, &) is in* EXPSPACE*;*
(2) INTERSECTION *for RE(#, &) is* PSPACE*-complete; and,*
(3) EQUIVALENCE *and* INCLUSION *for RE(#) is* EXPSPACE*-hard.*

Proof. (1) Follows directly from Theorem 5(1) and Theorem 6(1).

(2) The upper bound follows directly from Theorem 5(1) and Theorem 6(2). The lower bound is already known for ordinary regular expressions.

(3) This proof is provided in the body of the paper. □

Proof of Theorem 8:
INCLUSION *for CHARE(#) is* EXPSPACE*-complete.*

Proof. The EXPSPACE upper bound already follows from Theorem 7(1).

The proof for the EXPSPACE lower bound is in the same spirit as the proof for PSPACE-hardness of INCLUSION for CHAREs in [27].

The proof is a reduction from EXP-CORRIDOR TILING (see the proof of Theorem 7(3)). Thereto, let $T = (X, H, V, x_\perp, x_\top, n)$ be a tiling instance. Without loss of generality, we assume that $n \geq 2$. We construct two CHARE(#)-expressions r_1 and r_2 such that

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

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

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

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

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

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

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

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

$$r_1 = \overbrace{\$e\$e\$\cdots\$e\$}^{k \text{ times } e} \triangle x_\perp X^{[2^n-1,2^n-1]} \triangle X^*_\triangle \triangle x_\top X^{[2^n-1,2^n-1]} \triangle \overbrace{\$e\$e\$\cdots\$e\$}^{k \text{ times } e}; \text{ and,}$$
$$r_2 = \$X^*_{\triangle,\$}\$e_1\$e_2\$\cdots\$e_k\$X^*_{\triangle,\$}\$.$$

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

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

Notice that uwu' contains $2k+2$ times the symbol "\$". Moreover, the first and the last "\$" of uwu' is always matched onto the first and last "\$" of r_2. This means that $k+1$ consecutive \$-symbols of the remaining $2k$ \$-symbols in uwu' must be matched onto the \$-symbols in $\$e_1\$e_2\$\cdots\$e_k\$$. Hence, w is matched onto some e_i. So, w does not encode a correct tiling. As the sub-expression $\triangle x_\perp X^{[2^n-1,2^n-1]} \triangle X^*_\triangle \triangle x_\top X^{[2^n-1,2^n-1]} \triangle$ of r_1 defines all candidate tilings, the system T has no solution.

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

Proof of Theorem 9:

(1) INCLUSION *for CHARE(a, a?, a#) is* PSPACE *hard and in* EXPSPACE.
(2) EQUIVALENCE *for CHARE(a, a?, a#) is in* PTIME.
(3) INTERSECTION *for CHARE(a, a?, a#) is* NP*-complete.*

Proof. (1) The EXPSPACE upper bound is immediate from Theorem 7(1).

The proof for the PSPACE lower bound is in the same spirit as the proof of Theorem 8, except that we now use a reduction from CORRIDOR TILING instead

of EXP-CORRIDOR TILING, and we are no longer allowed to use the Kleene star operator.

The CORRIDOR TILING problem asks, given a tiling instance $T = (X, H, V, x_\bot, x_\top, n)$ whether there is a *correct corridor tiling for* T, that is, a mapping $\lambda : \{1, \ldots, m\} \times \{1, \ldots, n\} \to X$ for some m such that the first tiles of the first and last rows are $x_\bot$ and $x_\top$, respectively; and that the horizontal and vertical constraints are satisfied. Notice that the only difference between EXP-CORRIDOR TILING and CORRIDOR TILING is that the rows have length 2^n in the former problem, while they have length n in the latter problem.

Thereto, let Let $T = (X, H, V, x_\bot, x_\top, n)$ be a tiling instance. Without loss of generality, we assume that $n \geq 2$. We construct two CHARE$(a, a?, a\#)$-expressions r_1 and r_2 such that

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

As PSPACE is closed under complement, the PSPACE-hardness of INCLUSION for CHARE$(a, a?, a\#)$ follows.

Notice that there exists a correct corridor tiling for T if and only if there exists a correct corridor tiling for T with at most $|X|^n$ rows. Indeed, any correct corridor tiling with more than $|X|^n$ rows contains two times the same row and can be shortened due to a pumping argument.

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

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

in which each R_i represents a row and is in X^n. Moreover, R_0 is the bottom row and R_m is the top row. Let M be the number $|X|^n(n+1)+1$, which is the maximum length of the strings $(\dagger)$ that we need to consider. Let N be $2M + (|X|^2+1)(2|M|+n+3)$. The definition of N will become clear later in the proof. For the moment, it is only important to notice that we only need a polynomial number of bits for the binary representation of N. The following regular expressions detect strings of this form which do not encode a correct tiling for T:

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

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

– $X_{\triangle}^{[0,M]} \triangle X^{[0,n-1]} \triangle$,
– $X_{\triangle}^{[0,M]} \triangle X^{[n+1,M]} \triangle$,
– $X_{\triangle}^{[0,M]} x_1 x_2$, for every $(x_1, x_2) \in X^2 - H$,
– $X_{\triangle}^{[0,M]} x_1 X_{\triangle}^{[n-1,n-1]} x_2$, for every $(x_1, x_2) \in X^2 - V$, and
– $X_{\triangle}^{[0,M]}$.

Notice that the maximum length of a string in $L(e)$ is $(M+n+1)+(2M+2)+(|X|^2(M+2))+(|X|^2(M+n+1))+M$, which is equal to N. Because of the leading and trailing $X_{\triangle}^{[0,N]}$ expressions in each e_i, we have that $L(e) \subseteq L(e_i)$ for every $i = 1, \ldots, k$. We are now ready to define r_1 and r_2:

$$r_1 = \overbrace{\$e\$e\$\cdots\$e\$}^{k \text{ times } e} \triangle x_\perp X^{[n-1,n-1]} \triangle X_{\triangle}^{[0,M]} \triangle x_\top X^{[n-1,n-1]} \triangle \overbrace{\$e\$e\$\cdots\$e\$}^{k \text{ times } e}; \text{ and,}$$

$$r_2 = \$X_{\triangle,\$}^{[0,kN]} \$e_1\$e_2\$\cdots\$e_k\$X_{\triangle,\$}^{[0,kN]}\$.$$

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

We first show the implication from left to right. Thereto, let $L(r_1) \subseteq L(r_2)$. Let uwu' be an arbitrary string in $L(r_1)$ such that $u, u' \in L(\$e\$e\$\cdots\$e\$)$ and $w \in \triangle x_\perp X^{[n-1,n-1]} \triangle X_{\triangle}^{[0,M]} \triangle x_\top X^{[n-1,n-1]} \triangle$. Hence, $uwu' \in L(r_2)$.

Notice that uwu' contains $2k+2$ times the symbol "\$". Moreover, the first and the last "\$" of uwu' is always matched onto the first and last "\$" of r_2. This means that $k+1$ consecutive \$-symbols of the remaining $2k$ \$-symbols in uwu' must be matched onto the \$-symbols in $\$e_1\$e_2\$\cdots\$e_k\$$. Hence, w is matched onto some e_i. So, w does not encode a correct tiling. As the sub-expression $\triangle x_\perp X^{[n-1,n-1]} \triangle X_{\triangle}^{[0,M]} \triangle x_\top X^{[n-1,n-1]} \triangle$ of r_1 defines all candidate tilings, the system T has no solution.

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

(2) It is shown in [27] that two CHARE$(a, a?)$-expressions are equivalent if and only if they have the same *sequence normal form* (which is defined below). As $a^{[k,\ell]}$ is equivalent to $a^k(a?)^{\ell-k}$, we also have that two CHARE$(a, a?, a\#)$-expressions are equivalent if and only if they have the same sequence normal form.

We have to argue that the sequence normal form of CHARE$(a, a?, a\#)$-expressions can be computed in polynomial time. To this end, let $r = r_1 \cdots r_n$ be a CHARE$(a, a?, a\#)$-expression with factors $r_1, \ldots, r_n$. The *sequence normal form* of a CHARE$(a, a?)$ $r = r_1 \cdots r_n$ is obtained in the following way. First, we replace every factor of the form

– a by $a[1,1]$;
– $a?$ by $a[0,1]$; and,

– $a^{[k,\ell]}$ by $a[k,\ell]$.

where a is an alphabet symbol. We call a the *base symbol* of the factor $a[i,j]$. Then, we replace successive subexpressions $a[i_1,j_1]$ and $a[i_2,j_2]$ with the same base symbol a by $a[i_1+i_2, j_2+j_2]$ when j_1 and j_2 are integers until no such replacements can be made anymore. For instance, the sequence normal form of $aa?a^{2,5}a?bb?b?b^{[1,7]}$ is $a[3,8]b[2,10]$.

Obviously, the above algorithm to compute sequence normal form of CHARE(a, $a?, a\#$)-expressions can be implemented in polynomial time. It can be tested in linear time whether two sequence normal forms are the same.

(3) The NP-hardness of this problem is immediate since INTERSECTION is already NP-complete for CHARE($a, a?$)-expressions [27].

We show that the problem is in NP. To this end, we represent a string w by its sequence normal form, as defined in the proof of Theorem 9(2). We call such a string a *compressed string*. Let $r_1, \ldots, r_n$ be CHARE($a, a?, a\#$) expressions.

Lemma 15. *If $\bigcap_{i=1}^n L(r_i) \neq \emptyset$, then there exists a string $w = a_1^{j_1} \cdots a_m^{j_m} \in \bigcap_{i=1}^n L(r_i)$ such that $m \leq \min\{|r_i| \mid i \in \{1,\ldots,n\}\}$ and, for each $i = 1,\ldots,n$, j_i is not larger than the largest integer occurring in $r_1, \ldots, r_n$.*

Proof. Suppose that there exists a string $w = a_1^{j_1} \cdots a_m^{j_m} \in \bigcap_{i=1}^n L(r_i)$, with $a_i \neq a_{i+1}$ for every $i = 1, \ldots, m-1$. Since w is matched by every expression $r_1, \ldots, r_n$, and since a factor of a CHARE($a, a?, a\#$)-expression can never match a strict superstring of $a_i^{j_i}$ for $i = 1, \ldots, n$, we have that $m \leq \min\{|r_i| \mid i \in \{1,\ldots,n\}\}$.

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

The NP algorithm consists of guessing a compressed string w of polynomial size and verifying whether $w \in \bigcap_{i=1}^n L(r_i)$. To verify whether w is in the intersection, we essentially do the following. We start by representing w as a compressed string $a_1[i_1,i_1]\cdots a_m[i_m,i_m]$ and each regular expression by its sequence normal form. We then guess how w should be matched to each regular expression and we verify in polynomial time whether we have guessed this correctly.

To this end, let $r = b_1[k_1,\ell_1]\cdots b_{m_0}[k_{m_0},\ell_{m_0}]$ be a CHARE($a, a?, a\#$)-expression in sequence normal form. Formally, we guess a sequence of integers $j_1, \ldots, j_{m_0}$ such that, for each $i = 1, \ldots, m_0$, $k_i \leq j_i \leq \ell_i$. Each such integer j_i represents the number of symbols of w that will be matched with $b_i[k_i,\ell_i]$.

We describe a PTIME procedure to test whether $j_1, \ldots, j_{m_0}$ represents a correct match between w and r. We start by reading j_1.

(1) While there are still integers j_i left, do the following. Read j_i and verify whether the first j_i symbols of w can be matched onto $b_i[k_i,\ell_i]$. This is the case if and only if the first factor of w is of the form $b_i[x,x]$ with $x \leq j_i$.
(2) If the test in step (1) fails, reject.
(3) If the test in step (1) is successful, remove the j_i first symbols of w (that is, replace its first factor $b_i[x,x]$ of w by $b_i[x-j_i, x-j_i]$ or remove it when $j_i = x$. Return to step (1). □

Proof of Theorem 10:
INCLUSION, EQUIVALENCE, *and* INTERSECTION *for CHARE(*$a, a\#^{>0}$*) are in* PTIME.

Proof. The upper bound for EQUIVALENCE is immediate from Theorem 9(2).

For INCLUSION, let r_1 and r_2 be two CHARE($a, a\#^{>0}$) in sequence normal form. Let $r_1 = a_1[k_1, \ell_1] \cdots a_n[k_n, \ell_n]$ and $r_2 = a'_1[k'_1, \ell'_1] \cdots a'_{n'}[k'_{n'}, \ell'_{n'}]$. Notice that every number $k_1, \ldots, k_n, k'_1, \ldots, k'_{n'}$ is greater than zero. We claim that $L(r_1) \subseteq L(r_2)$ if and only if

- $n = n'$;
- for every $i = 1, \ldots, n$, $a_i = a'_i$;
- for every $i = 1, \ldots, n$, $k_i \geq k'_i$; and,
- for every $i = 1, \ldots, n$, $\ell_i \leq \ell'_i$.

Indeed, if $n \neq n'$, or if there exists an i such that $a_i \neq a'_i$ or $k_i < k'_i$, then $a_1^{k_1} \cdots a_n^{k_n} \in L(r_1) - L(r_2)$. If there exists an i such that $\ell_i > \ell'_i$, then $a_1^{\ell_1} \cdots a_n^{\ell_n} \in L(r_1) - L(r_2)$. Conversely, it is immediate that every string in $L(r_1)$ is also in $L(r_2)$. It is straightforward to test the four above conditions in linear time.

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

(i) $m_1 = m_2 = \cdots = m_n$;
(ii) for every $i, j = 1, \ldots, n$ and $x = 1, \ldots, m_1$, $a_{i,x} = a_{j,x}$; and,
(iii) for every $x = 1, \ldots, m_1$, $\max\{k_{i,x} \mid 1 \leq i \leq n\} \leq \min\{\ell_{i,x} \mid 1 \leq i \leq n\}$.

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

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

Proofs for Section 5

We provide some extra terminology for the proofs of Section 5.

We say that a DTD($\mathcal{M}$) (Σ, d, s), is reduced if, for every $a \in \Sigma$, there exists a tree $t \in L(d)$ such that a is a label in t.

We say that an EDTD($\mathcal{M}$) $D = (\Sigma, \Sigma', d, s, \mu)$ is *reduced* if the DTD (Σ', d, s) is reduced. *Reducing* an EDTD D is the act of finding an equivalent reduced EDTD.

In the proofs of the following propositions, we will use the fact that reducing an EDTD(#,&) is tractable.

Lemma 16. *Reducing a DTD(#,&) is in polynomial time.*

Proof. The algorithm works along the same lines as for tree automata but is slightly more involved due to the RE(#, &) expressions. First, it computes in a bottop-up pass which symbols do not generate a tree and removes these symbols from the DTD. Then it computes in a top-down pass which symbols are not reachable from the start symbol and removes these symbols from the DTD.

However, we need to take a little bit of care about the RE(#, &)-expressions. Given a RE(#, &)-expression r and a set of alphabet symbols S, we need to be able to construct an RE(#, &) expression for $L(r) \cap S^*$. We do this as follows. First, replace every symbol in $\Sigma - S$ occurring in r by $\emptyset$. Now, we eliminate all symbols $\emptyset$ from r by applying the following rules until no rule can be applied anymore:

- replace $\emptyset \cdot a$ or $a \cdot \emptyset$ by $\emptyset$;
- replace $\emptyset^*$ by ε;
- replace $(\emptyset + r')$ or $(r' + \emptyset)$ by r';
- replace $(\emptyset \,\&\, r')$ or $(r' \,\&\, \emptyset)$ by $\emptyset$;
- replace $\emptyset^{[k,\ell]}$ by $\emptyset$ if $k > 0$; and
- replace $\emptyset^{[k,\ell]}$ by ε if $k = 0$,

where r' is a subexpression of r. If we extend the semantics of RE(#, &) expressions in the straightforward manner to RE(#, &) expressions with $\emptyset$ (where $\emptyset$ is the symbol defining the empty language), then it is easy to see that the above rules only replace subexpressions of r by expressions that are equivalent. Moreover, since every rule either eliminates an $\emptyset$ or a subexpression of r, we can only apply a linear number of rewrite rules to a given RE(#, &) expression r. Hence, we can rewrite r in quadratic time.

Let (Σ, d, s) be a DTD(#,&). We reduce d as follows.

1. Compute $R_1 = \{a \in \Sigma \mid L((\Sigma, d, a)) \neq \emptyset\}$. This can be done in the standard bottom-up manner: let $R_{1,1} := \{a \mid \varepsilon \in L(d(a))\}$ and, for every $i = 1, \ldots, |\Sigma| - 1$, let $R_{1,i+1} := \{a \mid L(d(a)) \cap L((R_i)^*) \neq \emptyset\}$. Then, $R_1 = R_{1,|\Sigma|}$.
2. If $R_1 = \emptyset$, then return the empty DTD. Otherwise, let $(\Sigma - R_1, d_1, s)$ be obtained from d by replacing every RE(#, &)-expression r in d by an RE(#, &) expression for $L(r) \cap (\Sigma - R_1)^*$.
3. Compute the *reachable* symbols R_2 of d_1. That is, s is reachable and, if a is reachable and there is a string $w_1 b w_2 \in L(d(a))$, then b is also reachable. Computing R_2 is straightforward.
4. Let $(\Sigma - (R_1 \cup R_2), d_2, s)$ be obtained from d_1 by replacing every RE(#, &)-expression r in d_1 by an RE(#, &) expression for $L(r) \cap (\Sigma - (R_1 \cup R_2))^*$.

The DTD $(\Sigma - (R_1 \cup R_2), d_2, s)$ is a reduced DTD which is equivalent to d. The above algorithm computes d_2 in polynomial time. □

Corollary 17. *Reducing an EDTD(#,&) is in polynomial time.*

Proof of Proposition 11:
Let $\mathcal{R}$ be a subclass of RE(#, &) and let $\mathcal{C}$ be a complexity class which contains PTIME *and is closed under positive reductions. Then the following are equivalent:*

(a) INCLUSION *for* $\mathcal{R}$ *expressions is in* $\mathcal{C}$.
(b) INCLUSION *for* $DTD(\mathcal{R})$ *is in* $\mathcal{C}$.
(c) INCLUSION *for* $EDTD^{st}(\mathcal{R})$ *is in* $\mathcal{C}$.

The corresponding statement holds for EQUIVALENCE.

Proof. It suffices to prove the implication from (a) to (c). Thereto, let $D_1 = (\Sigma, \Sigma'_1, d_1, s_1, \mu_1)$ and $D_2 = (\Sigma, \Sigma'_2, d_2, s_2, \mu_2)$ be two single-type EDTDs. We can assume $\Sigma'_1 \cap \Sigma'_2 = \emptyset$. We first need to reduce D_1 and D_2, which can be done in polynomial time, according to Corollary 17.

We then compute a correspondence relation $\sim \subseteq \Sigma'_1 \times \Sigma'_2$ as follows:

- $s_1 \sim s_2$; and,
- if $\tau_1 \sim \tau_2$, then for every $a \in \Sigma$, $\tau'_1 \sim \tau'_2$ where τ'_1 is the unique a-type in $d_1(\tau_1)$ and τ'_2 is the unique a-type in $d_2(\tau_2)$.

It now can be shown that $L(D_1) \subseteq L(D_2)$ iff for every $\tau_1 \in \Sigma'_1$ and $\tau_2 \in \Sigma'_2$ with $\tau_1 \sim \tau_2$, $L(\mu_1(d_1(\tau_1))) \subseteq L(\mu_2(d_2(\tau_2)))$. Notice that, because of the single-type property, $d_i(\tau_i)$ is an $\mathcal{R}$ expression iff $\mu_i(d_i(\tau_i))$ is an $\mathcal{R}$ expression for each $i = 1, 2$. □

Proof of Theorem 13:

(1) EQUIVALENCE *and* INCLUSION *for EDTD(#,&) is in* EXPSPACE*;*
(2) EQUIVALENCE *and* INCLUSION *for EDTD(#) and EDTD(&) is* EXPSPACE*-hard;*
(3) INTERSECTION *for EDTD(#,&) is* EXPTIME*-complete.*

Proof. (1) We show that INCLUSION is in EXPSPACE. The upper bound for EQUIVALENCE then immediately follows.

First, we introduce some notation. For an EDTD $D = (\Sigma, \Sigma', d, s, \mu)$, we will denote elements of Σ', i.e., types, by τ. We denote by (D, τ) the EDTD D with start symbol τ. We define the *depth* of a tree t, denoted by $\text{depth}(t)$, as follows: if $t = \varepsilon$, then $\text{depth}(t) = 0$; and if $t = \sigma(t_1 \cdots t_n)$, then $\text{depth}(t) = \max\{\text{depth}(t_i) \mid i = 1, \ldots, n\} + 1$.

Suppose that we have two EDTDs $D_1 = (\Sigma, \Sigma'_1, d_1, s_1, \mu_1)$ and $D_2 = (\Sigma, \Sigma'_2, d_2, s_2, \mu_2)$. We provide an EXPSPACE algorithm that decides whether $L(D_1) \not\subseteq L(D_2)$. As EXPSPACE is closed under complement, the theorem follows. The algorithm computes a set E of pairs $(C_1, C_2) \in 2^{\Sigma'_1} \times 2^{\Sigma'_2}$ where $(C_1, C_2) \in E$ iff there exists a tree t such that $C_j = \{\tau \in \Sigma'_j \mid t \in L((D_j, \tau))\}$ for each $j = 1, 2$. That is, every C_j is the set of types that can be assigned by D_j to the root of t. Or when viewing D_j as a tree automaton, C_j is the set of states that can be assigned to the root in a run on t. Therefore, we say that t *is a witness for* (C_1, C_2). Notice that $t \in L(D_1)$ (resp., $t \in L(D_2)$) if $s_1 \in C_1$ (resp. $s_2 \in C_2$). Hence, $L(D_1) \not\subseteq L(D_2)$ iff there exists a pair $(C_1, C_2) \in E$ with $s_1 \in C_1$ and $s_2 \notin C_2$.

We compute the set E in a bottom-up manner as follows:

1. Initially, set $E_1 := \{(C_1, C_2) \mid \exists \tau_1 \in \Sigma_1', \tau_2 \in \Sigma_2' \text{ such that } \mu_1(\tau_1) = \mu_2(\tau_2), C_1 = \{\tau_1 \in \Sigma_1' \mid \varepsilon \in d_1(\tau_1)\}, \text{ and } C_2 = \{\tau_2 \in \Sigma_2' \mid \varepsilon \in d_2(\tau_2)\}\}$.
2. For every $k > 1$, E_k is the union of E_{k-1} and the pairs (C_1, C_2) for which there is an $a \in \Sigma$ and a string $(C_{1,1}, C_{2,1}) \cdots (C_{1,n}, C_{2,n})$ in E_{k-1}^* such that

$$C_j = \{\tau \in \Sigma_j' \mid \mu_j(\tau) = a, \exists b_{j,1} \in C_{j,1}, \ldots, b_{j,n} \in C_{j,n} \text{ with } b_{j,1} \cdots b_{j,n} \in d_j(\tau)\},$$

   for each $j = 1, 2$.
3. $E := E_\ell$ for $\ell = 2^{|\Sigma_1'|} \cdot 2^{|\Sigma_2'|}$.
4. Accept when there is a pair $(C_1, C_2) \in E$ with $s_1 \in C_1$ and $s_2 \notin C_2$. Reject otherwise.

We argue that the algorithm is correct. As $E_k \subseteq E_{k+1}$, for every k, it follows that $E_\ell = E_{\ell+1}$. Hence, the algorithm computes the largest set of pairs. The following lemma then shows that the algorithm decides whether $L(D_1) \not\subseteq L(D_2)$. The lemma can be proved by induction on k.

Lemma 18. *For every $k \geq 1$, $(C_1, C_2) \in E_k$ if and only if there exists a witness tree for (C_1, C_2) of depth at most k.*

It remains to show that the algorithm can be carried out using exponential space. Step (a) reduces to a linear number of tests $\varepsilon \in L(r)$, for some RE(#, &)-expressions r which is in PTIME by [20]. Step (c) and (d) can be carried out in exponential time, since the size of E is exponential in the input. For step (b), it suffices to argue that, when E_{k-1} is known, it is decidable in EXPSPACE whether a pair (C_1, C_2) is in E_k. As there are only an exponential number of such possible pairs, the result follows. To this end, we need to verify that there exists a string $W = (C_{1,1}, C_{2,1}) \cdots (C_{1,n}, C_{2,n})$ in E_{k-1}^* such that for each $j = 1, 2$,

(A) for every $\tau \in C_j$, there exist $b_{j,1} \in C_{j,1}, \ldots, b_{j,n} \in C_{j,n}$ with $b_{j,1} \cdots b_{j,n} \in d_j(\tau)$; and,
(B) for every $\tau \in \Sigma_j' \setminus C_j$, there do *not* exist $b_{j,1} \in C_{j,1}, \ldots, b_{j,n} \in C_{j,n}$with $b_{j,1} \cdots b_{j,n} \in d_j(\tau)$.

Assume that $\Sigma_1' \cap \Sigma_2' = \emptyset$. Let, for each $j = 1, 2$ and $\tau \in \Sigma_j'$, $N(\tau)$ be the NFA(#, &) accepting $d_j(\tau)$. Intuitively, we guess the string W one symbol at a time and compute the set of reachable configurations Γ_τ for each $N(\tau)$.

Initially, Γ_τ is the singleton set containing the initial configuration of $N(\tau)$. Suppose that we have guessed a prefix $(C_{1,1}, C_{2,1}) \cdots (C_{1,m-1}, C_{2,m-1})$ of W and that we guess a new symbol $(C_{1,m}, C_{2,m})$. Then, we compute the set $\Gamma_\tau' = \{\gamma' \mid \exists b \in C_{j,m}, \gamma \in \Gamma_\tau \text{ such that } \gamma \Rightarrow_{N(\tau),b} \gamma'\}$ and set Γ_τ to Γ_τ'.Each set Γ_τ' can be computed in exponential space from Γ_τ. We accept (C_1, C_2) when for every $\tau \in \Sigma_j'$, $\tau \in C_j$ iff Γ_τ contains an accepting configuration.

(2) It is shown by Mayer and Stockmeyer that EQUIVALENCE and INCLUSION are EXPSPACE-hard for RE(#, &)s. Hence, EQUIVALENCE and INCLUSION are also EXPSPACE hard for EDTD(&). Hardness for EDTD(#) is immediate from Theorem 7(2).

(3) The lower bound follows from [36]. We argue that the problem is in EXPTIME. Thereto, let, for each $i = 1, \ldots, n$, $D_i = (\Sigma, \Sigma'_i, d_i, s_i, \mu_i)$ be an EDTD(#,&). We assume w.l.o.g. that the sets Σ'_i are pairwise disjoint. We also assume that the start type s_i never appears at the right-hand side of a rule. Finally, we assume that no derivation tree consists of only the root. For each type $\tau \in \Sigma'_i$, let $N(\tau)$ denote an NFA(#, &) for $d_i(\tau)$. According to Theorem 5, $N(\tau)$ can be computed from $d_i(\tau)$ in polynomial time. We provide an alternating polynomial space algorithm that guesses a tree t and accepts if $t \in L(D_1) \cap \cdots \cap L(D_n)$. As APSPACE = EXPTIME, this shows the theorem.

We guess t node by node in a top-down manner. For every guessed node v, the following information is written on the tape of the TM: for every $i \in \{1, \ldots, n\}$, the triple $c_i = (\tau^i_v, \tau^i_p, \gamma^i)$ where τ^i_v is the type assigned to v by grammar D_i, τ^i_p is the type of the parent assigned by D_i, and γ^i is the current configuration $N(\tau^i_p)$ is in after reading the string formed by the left siblings of v. In the following, we say that $\tau \in \Sigma'_i$ is an a-type when $\mu_i(\tau) = a$.

The algorithms proceeds as follows:

1. As for each grammar the types of the roots are given, we start by guessing the first child of the root. That is, we guess an $a \in \Sigma$, and for each $i \in \{1, \ldots, n\}$, we guess a type τ^i and write the triple $c_i = (\tau^i, s_i, \gamma^i_s)$ on the tape where γ^i_s is the start configuration of $N(s_i)$.
2. For $i \in \{1, \ldots, n\}$, let $c_i = (\tau^i, \tau^i_p, \gamma^i)$ be the triples on the tape. The algorithm now universally splits into two parallel branches as follows:
   (a) **Downward extension**: When for every i, $\varepsilon \in d_i(\tau^i)$ then the current node can be a leaf node and the branch accepts. Otherwise, guess an $a \in \Sigma$ and for each i, guess an a-type θ^i. Replace every c_i by the triple $(\theta^i, \tau^i, \gamma^i_s)$ and proceed to step (b). Here, γ^i_s is the start configuration of $N(\tau^i)$.
   (b) **Extension to the right**: For every $i \in \{1, \ldots, n\}$, compute a configuration γ'^i for which $\gamma^i \Rightarrow_{N(\tau^i_p), \tau^i} \gamma'^i$. When every γ^i is a final configuration, then we do not need to extend to the right anymore and the algorithm accepts. Otherwise, guess an $a \in \Sigma$ and for each i, guess an a-type θ^i. Replace every c_i by the triple $(\theta^i, \tau^i, \gamma'^i)$ and proceed to step (b).

We argue that the algorithm is correct. If the algorithm accepts, we have guessed a tree t and, for every $i = 1, \ldots, n$, a tree t'_i with $\mu_i(t'_i) = t$ and $t'_i \in L(d_i)$. Therefore, $t \in \bigcap_{i=1}^n L(D_i)$. For the other direction, suppose that there exists a tree $t \in \bigcap_{i=1}^n L(D_i)$ and t is minimal in the sense that no subtree t_0 of t is in $\bigcap_{i=1}^n L(D_i)$. Then, there is a run of the above algorithm that guesses t and guesses trees t'_i with $\mu_i(t'_i) = t$. The tree t must be minimal since the algorithm stops extending the tree as soon as possible.

The algorithm obviously uses only polynomial space.

□

Proofs for Section 6

We need a bit of terminology for the proof of Theorem 14. Let t be a tree and v be a node. By anc-str$^t(v)$ we denote the string formed by the labels on the path from the root to v, i.e., $\text{lab}^t(\varepsilon)\text{lab}^t(i_1)\text{lab}^t(i_1i_2)\cdots\text{lab}^t(i_1i_2\cdots i_k)$ where $v = i_1i_2\cdots i_k$.

We say that a tree language L is *closed under ancestor-guarded subtree exchange* if the following holds. Whenever for two trees $t_1, t_2 \in L$ with nodes $u_1 \in \text{Dom}(t_1)$ and $u_2 \in \text{Dom}(t_2)$ it holds that anc-str$^{t_1}(u_1) = $ anc-str$^{t_2}(u_2)$ implies $t_1[u_1 \leftarrow \text{subtree}^{t_2}(u_2)] \in L$. Here, $t_1[u_1 \leftarrow \text{subtree}^{t_2}(u_2)]$ denotes the tree obtained from t_1 by replacing its subtree rooted at u_1 by the subtree rooted at u_2 in t_2.

We recall the following theorem from [28]:

Theorem 19 (Theorem 7.1 in [28]). *Let L be a tree language defined by an EDTDs. Then the following conditions are equivalent.*

(a) T is definable by a single-type EDTD.
(b) T is closed under ancestor-guarded subtree exchange.

We are now ready for the proof of Theorem 14.

Proof of Theorem 14:
Given an EDTD(#,&), deciding whether it is equivalent to an EDTD$^{st}(\#,\&)$ or DTD(#,&) is EXPSPACE*-complete.*

Proof. We first show that the problem is hard for EXPSPACE. We use a reduction from universality of RE$(\#,\&)$, i.e., deciding whether an RE$(\#,\&)$-expression is equivalent to Σ^*. The proof of Theorem 7(2) shows that the latter is EXPSPACE-hard.

To this end, let r be an RE$(\#,\&)$-expression over Σ and let b and s be two symbols not occurring in Σ. By definition $L(r) \neq \emptyset$. Define $D = (\Sigma \cup \{b, s\}, \Sigma \cup \{s, b^1, b^2\}, d, s, \mu)$ as the EDTD with the following rules:

$$\begin{aligned} s &\to (b^1)^* b^2 (b^1)^* \\ b^1 &\to \Sigma^* \\ b^2 &\to r, \end{aligned}$$

where for every $\tau \in \Sigma \cup \{s\}$, $\mu(\tau) = \tau$, and $\mu(b^1) = \mu(b^2) = b$. We claim that D is equivalent to a single-type DTD or a DTD iff $L(r) = \Sigma^*$. Clearly, if r is equivalent to Σ^*, then D is equivalent to the DTD (and therefore also to a single-type EDTD)

$$\begin{aligned} s &\to b^* \\ b &\to \Sigma^*. \end{aligned}$$

Conversely, suppose that there exists an EDTD$^{\text{st}}$ which defines the language $L(D)$. Towards a contradiction, assume that r is not equivalent to Σ^*. Let w_r be a string in $L(r)$ and let $w_{\neg r}$ be a Σ-string not in $L(r)$. Consider the trees

$t_1 = s(b(w_r)b(w_{\neg r}))$ and $t_2 = s(b(w_{\neg r})b(w_r))$. Clearly, t_1 and t_2 are in $L(D)$. However, the tree $t = s(b(w_{\neg r})b(w_{\neg r}))$ obtained from t_1 by replacing its left subtree by the left subtree of t_2 is not in $L(D)$. According to Theorem 19, every tree language defined by a single-type EDTD is closed under such an exchange of subtrees. So, this means that $L(D)$ cannot be defined by an EDTD$^{\text{st}}$, which leads to the desired contradiction.

For the upper bound, we proceed as follows. Let $D = (\Sigma, \Sigma', d, s, \mu)$ be an EDTD. Intuitively, we compute an EDTD$^{\text{st}}$ $D_0 = (\Sigma, \Sigma'_0, d_0, s, \mu_0)$ which is the closure of D under the single-type property. The EDTD$^{\text{st}}$ D_0 has the following properties:

(a) Σ'_0 is in general exponentially larger than Σ';
(b) the RE$(\#, \&)$-expressions in the definition of d_0 are only polynomially larger than the RE$(\#, \&)$-expressions in the definition of d;
(c) $L(D) \subseteq L(D_0)$; and,
(d) $L(D_0) = L(D) \Leftrightarrow$ SIMPLIFICATION is true for D.

Hence, we have that SIMPLIFICATION is true for D if and only if $L(D_0) \subseteq L(D)$.

We first show how D_0 can be constructed. According to Corollary 17, we can assume w.l.o.g. that, for each type $a^i \in \Sigma'$, there exists a tree $t' \in L(d)$ such that a^i is a label in t'. For a string $w \in \Sigma^*$ and $a \in \Sigma$ let types(wa) be the set of all types $a^i \in \Sigma'$, for which there is a tree t and a tree $t' \in L(d)$ with $\mu(t') = t$, and a node v in t such that anc-str$^t(v) = wa$ and the type of v in t_0 is a^i. We show how to compute types(wa) in exponential time. To this end, we enumerate all sets types(w). Let $s = c^1$. Initially, set $W := \{c\}$, Types$(c) := \{c^1\}$ and $R := \{\{c^1\}\}$. Repeat the following until W becomes empty:

(1) Remove a string wa from W.
(2) For every $b \in \Sigma$, let Types(wab) contain all b^i for which there exists an a^j in Types(wa) and a string in $d(a^j)$ containing b^i. If Types(wab) is not empty and not already in R, then add it to R and add wab to W.

Since we add every set only once to R, the algorithm runs in time exponential in the size of D. Moreover, we have that Types$(w) = \mathit{types}(w)$ for every w, and that $R = \Sigma'_0$.

For each $a \in \Sigma$, let types(D, a) be the set of all nonempty sets types(wa), with $w \in \Sigma^*$. Clearly, each types(D, a) is finite. We next define $D_0 = (\Sigma, \Sigma'_0, d_0, s, \mu_0)$. Its set of types is $\Sigma'_0 := \bigcup_{a \in \Sigma} \text{types}(D, a)$. Note that $s \in \Sigma'_0$. For every $\tau \in \text{types}(D, a)$, set $\mu_0(\tau) = a$. In d_0, the right-hand side of the rule for each types(wa) is the disjunction of all $d(a^i)$ for $a^i \in \text{types}(wa)$, with each b^j in $d(a^i)$ replaced by types(wab).

We show that properties (a)–(d) hold. Since $\Sigma'_0 \subseteq 2^{\Sigma'}$, we immediately have that (a) holds. The RE$(\#, \&)$-expressions that we constructed in D_0 are unions of a linear number of RE$(\#, \&)$-expressions in D, but have types in $2^{\Sigma'}$ rather than in Σ'. Hence, the size of the RE$(\#, \&)$-expressions in D_0 is at most quadratic in the size of D. Finally, we note that it has been shown in Theorem 7.1 in [28] that (c) and (d) also hold.

It remains to argue that it can be decided in EXPSPACE that $L(D_0) \subseteq L(D)$. A direct application of the EXPSPACE algorithm in Theorem 13(2) leads to a 2EXPSPACE algorithm to test whether $L(D_0) \subseteq L(D)$, due to the computation of C_1. Indeed, the algorithm remembers, given the EDTDs $D_0 = (\Sigma, \Sigma'_0, d_0, s_0, \mu_0)$ and $D = (\Sigma, \Sigma', d, s, \mu)$, all possible pairs (C_1, C_2) such that there exists a tree t with $C_1 = \{\tau \in \Sigma'_0 \mid t \in L((D_0, \tau))\}$ and $C_2 = \{\tau \in \Sigma' \mid t \in L((D, \tau))\}$. It then accepts if there exists a such a pair (C_1, C_2) with $s_0 \in C_1$ and $s \notin C_2$. However, when we use non-determinism, notice that it is not necessary to compute the entire set C_1. Indeed, as we only test whether there *exist* elements in C_1 in the entire course of the algorithm (i.e. in steps 2 and 4), we can adapt the algorithm to compute pairs (c_1, C_2), where c_1 is an element of C_1, rather than the entire set. Since NEXPSPACE = EXPSPACE, we can use this adaption to test whether $L(D_0) \subseteq L(D)$ in EXPSPACE. □