Schema Design for XML Repositories: Complexity and Tractability *

Wim Martens†
Technical University of Dortmund

Matthias Niewerth
Technical University of Dortmund

Thomas Schwentick
Technical University of Dortmund

ABSTRACT

Abiteboul et al. initiated the systematic study of distributed XML documents consisting of several logical parts, possibly located on different machines. The physical distribution of such documents immediately raises the following question: how can a global schema for the distributed document be broken up into local schemas for the different logical parts? The desired set of local schemas should guarantee that, if each logical part satisfies its local schema, then the distributed document satisfies the global schema.

Abiteboul et al. proposed three levels of desirability for local schemas: local typing, maximal local typing, and perfect local typing. Immediate algorithmic questions are: (i) given a typing, determine whether it is local, maximal local, or perfect, and (ii) given a document and a schema, establish whether a (maximal) local or perfect typing exists. This paper improves the open complexity results in their work and initiates the study of (i) and (ii) for schema restrictions arising from the current standards: DTDs and XML Schemas with deterministic content models. The most striking result is that these restrictions yield tractable complexities for the perfect typing problem.

Furthermore, an open problem in Formal Language Theory is settled: deciding language primality for deterministic finite automata is PSPACE-complete.

Categories and Subject Descriptors

H.2.1 [**Database Management**]: Logical Design; H.2.3 [**Database Management**]: Languages—*Data description languages (DDL)*; H.2.4 [**Database Management**]: Systems—*Distributed Databases*; F.4.3 [**Mathematical Logic and Formal Languages**]: Formal Languages

General Terms

Algorithms, Design, Theory

Keywords

XML, XML schemas, complexity, language primality

1. INTRODUCTION

Information has become more and more distributed since the arrival of the Web. The distribution of XML data is even essential in many areas such as e-commerce, collaborative editing, or network directories [1]. When dealing with such distributed XML data, it is desirable to have a system that can grant a large amount of independence to individual peers, while at the same time also being able to deal with the data as a whole. One way of achieving this could be through *XML repositories*. An XML repository is a collection of XML documents, together with a set of tools for manipulating and validating these documents.

In this paper we focus on an important part of such XML repositories, namely on the collection of XML documents and on schema design for such a collection. We abstract collections of XML documents as *distributed XML documents*. Following Abiteboul et al. [1], a distributed XML document consists of a *root document* T, which is an XML tree that is stored locally at some site. Some of the leaves of T are labeled with references to external resources, say $f_1, \ldots, f_n$. The extension $\text{ext}(T)$ of T is then obtained by replacing each node f_i with the XML tree or XML forest provided by the resource r_i referenced by f_i. In other words, $\text{ext}(T)$ is a large XML document that is distributed into several logical parts. The root document T provides an interface to this large XML document and has through its pointers f_i the knowledge of where to get access to the different parts. These parts can be maintained by different peers and/or provided by programs or web service calls. We therefore sometimes also refer to the f_i as *function calls*.

We provide a very simple example to illustrate these concepts. Suppose that we want to run a web site from which we can query auction data from various on-line auction platforms in different countries. The idea is that users would be able to search for keywords on this web site and that they can investigate, e.g., for how much money items with a matching description have been sold internationally. Such a web site could, for example, combine data from the various national

*We acknowledge the financial support of the Future and Emerging Technologies (FET) programme within the Seventh Framework Programme for Research of the European Commission, under the FET-Open grant agreement FOX, number FP7-ICT-233599

†Supported by a grant of the North-Rhine Westfalian Academy of Sciences and Arts, and the Stiftung Mercator Essen.

PODS'10, June 6–11, 2010, Indianapolis, Indiana, USA.

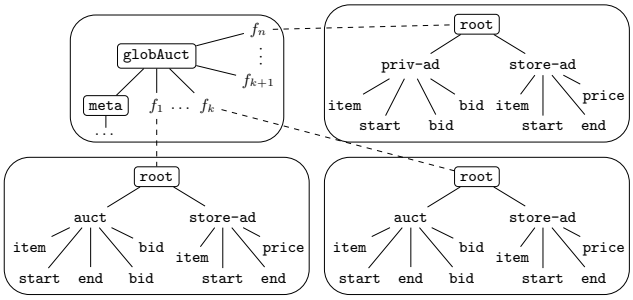


Figure 1: A distributed document. The root document T is in the upper left block.

Ebay and Yahoo auction web sites, and from the plethora of platforms with second-hand ads around the world. Figure 1 contains a vastly simplified possible distributed XML document for such a scenario and Figure 2 contains its extension. This document gathers three kinds of advertisements: (1) auctions, which have a certain start and end time, and on which people may bid, (2) store advertisements, which are advertisements for items that stores offer on-line and can be bought for a fixed price, and (3) private advertisements, which are advertisements for, e.g., second-hand items that people sell, without a predefined end-date, and on which people may freely bid.

A global schema S for ext(T) could be given by the following Document Type Definition (DTD) D:

```
<!ELEMENT globAuct (meta,(auct|store-ad|priv-ad)+)>
<!ELEMENT auct (item,start,end,bid*)>
<!ELEMENT store-ad (item,start,end,price)>
<!ELEMENT priv-ad (item,start,bid*)>
<!ELEMENT meta (...)>
```

For each element in an XML document, such a DTD specifies the allowed labels of its children. For example, the first rule of the DTD states that the labels of the sequence of children of each `globAuct` element must adhere to the regular expression `meta,(auct|store-ad|priv-ad)+`. We use the term *content model* of an element to refer to its associated regular expression in the DTD.

We recall the problems defined in [1]. Given a global schema S and a distributed XML document T, we want to know whether we can design schemas for $r_1, \ldots, r_n$ such that each r_i has as much freedom as possible and is independent of the other r_i's. We would like to provide each r_i with an XML schema S_i such that, (1) if each r_i validates its document against S_i, then ext(T) satisfies S (soundness) and (2) we do not introduce more restrictions than this global type (completeness). Such a tuple $(S_1, \ldots, S_n)$ is called a *typing*. A typing that is sound and complete is called a *local typing*. When a local typing gives a maximum degree of freedom to each individual peer, it is called a *maximal local typing*. A typing is *perfect* if it is the unique maximal typing for all sound typings.

It is easy to see that there is a sound typing for the DTD D above and the distributed document of Figure 1: we provide each r_i with the DTD D_{auct}:

```
<!ELEMENT root (auct|store-ad|priv-ad)+>
<!ELEMENT auct (item,start,end,bid*)>
<!ELEMENT store-ad (item,start,end,price)>
<!ELEMENT priv-ad (item,start,bid*)>
```

This design, however, is overly restrictive. We only need one resource with DTD D_{auct} to provide an element, for example. We could have assigned `(auct|store-ad|priv-ad)+` to one peer and `(auct|store-ad|priv-ad)*` to all others, which would even be a maximal local typing. However, this typing is, in a sense, unfair: we are restricting one peer, whereas there are other possibilities (other maximal local typings) in which this peer is not restricted. On the other hand, if the content model for `globAuct` in D would have been `(meta,(auct|store-ad|priv-ad)*)`, then even a perfect typing would have been possible. We show in this paper, given a distributed document and a schema S, how to compute a perfect typing in polynomial time, if it exists.[1]

For an example as simple as the one illustrated here it is easy to see how a design with the desirable properties can be constructed. However, the various standards allow almost[2] arbitrary regular expressions for defining content models and therefore XML schemas and designs of distributed documents can become much more complex. The reason why a global design is bad can therefore be much more subtle than can be inferred from our simple example. We believe that, in such real-life scenarios, the techniques presented in this paper can be of a great help to a human designer.

From XML Documents to Strings. Abiteboul et al. studied the typing problems for DTDs ([5], Section 2.8) XML Schemas [11], and extended DTDs [17] as schema languages. Given a typing S for a design (D, T), where D is either a DTD, XML Schema, or extended DTD and T is a distributed XML document, they study LOC(DTD), ML(DTD), and PERF(DTD). These problems ask whether S is a local, maximal local, or perfect typing for D and T, respectively. They also studied the problems $\exists$−LOC(DTD), $\exists$−ML(DTD), and $\exists$−PERF(DTD), where only the design is given and the question is whether a local, maximal local, or perfect typing exists.

It is known that several decision problems for DTDs and XML Schemas can be reduced to corresponding problems on strings. For example, in [16] it is shown that containment and equivalence testing for DTDs and XML Schemas has the same complexity as containment and equivalence testing for the (classes of) regular expressions that these DTDs and XML Schemas use. This result was extended by [1] in the context of perfect and (maximal) local typings. In this sense, it follows from [1, 16] that all the abovementioned problems have the same complexity for DTDs as for the regular expressions that these DTDs use. For this reason, as long as we are interested in DTD and XML Schema, we can safely focus our study to strings instead of trees. In other words, we study designs (D, T) where D is a regular expression or finite automaton, and T is a *distributed string* $w_0 f_1 w_1 \cdots f_n w_n$ with function calls $f_1, \ldots, f_n$ and $w_0, \ldots, w_n$ are words from a finite alphabet Σ.[3] It should be noted that the typing problems for Relax NG schemas [8] or extended DTDs cannot be reduced to the string case if EXPTIME $\neq$ PSPACE.

Our Contributions. By NFA, DFA, and DRE, we ab-

[1] Abiteboul et al. proved a tight PSPACE bound for the perfect typing problem if non-deterministic content models are allowed. Since the World Wide Web Consortium requires content models to be *deterministic*, we focus on deterministic content models.

[2] I.e., *deterministic*.

[3] Distributed strings are formally defined in Section 2.

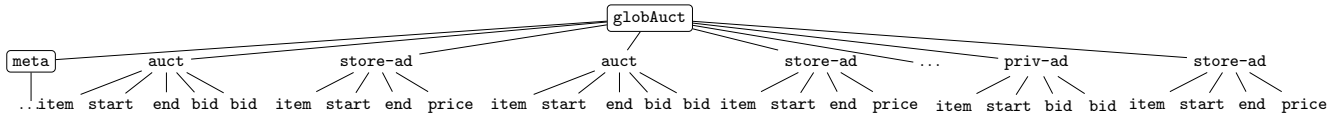


Figure 2: Extension ext(T) for the distributed document in Figure 1.

breviate non-deterministic finite automata, deterministic finite automata, and deterministic regular expressions, respectively. The contributions in this paper are the following.

- We connect the schema design questions with language equations in Formal Language Theory. Through this connection we can derive that the questions whether local or maximal local typings exist coincide when content models are given by NFAs or DFAs. Whenever a (possibly non-regular) local typing exists, there also exists a regular *maximal* local typing (Section 2.4). However, this fails for DREs.

- We provide new structural insights into the perfect typing problem, by giving a new construction of a *perfect automaton* for DFAs. This automaton admits to infer PTIME algorithms for all perfect typing problems with deterministic content models (Section 3).

- We provide normal forms for local and maximal local typings, that reduce the search space for typings and significantly improve the upper bounds on the local and maximal local typing problems. For example, we improve the upper bound for the computationally most difficult problem, $\exists-$LOC(NFA), from 2EXPSPACE to EXPSPACE (Section 4).

- With respect to testing of properties of typings, we pinpoint all complexities; all problems are either in PTIME or PSPACE-complete. One would expect the complexity of the typing problems to drop significantly when going from NFAs to DFAs or DREs. Remarkably, only the perfect typing problems become tractable and the (maximal) local problems remain PSPACE-complete (Section 5).

- In the existential setting of the problem, we show that the specific case where the distributed string has two function calls is essentially equivalent to the Primality problem in Formal Languages: Given a deterministic finite automaton A, determine whether there exist two regular languages L_1, L_2 such that $L(A) = L_1 \cdot L_2$, where $L_1 \neq \{\varepsilon\} \neq L_2$. The complexity of Primality has been an open problem in Formal Language Theory since the late 90's (see, e.g., [19, 10, 2, 13, 18, 20, 23]). Beyond sheer decidability no further upper or lower complexity bounds were known [20].[4] We prove that Primality is PSPACE-complete and we settle the complexity of $\exists-$LOC(DFA) for distributed strings with at most two function calls (Section 6).

- Intuitively, one may think that the typing problems for DFAs are often the same as for DREs. We prove that these problems are different in almost all cases.

[4]Salomaa mentions the complexity of Primality as Problem 2.1 in his recent survey [20].

As we mentioned before, all our upper and lower bounds for NFAs, DFAs, and DREs also hold for DTDs and XML Schemas that use NFAs, DFAs, and DREs as formalism for their content models, respectively.

We would like to stress the cross-fertilization between formal languages and database theory in this work: we drew inspiration from the work on language equations for our setting of distributed XML, and distributed XML typing questions helped to find a solution to a question in Formal Languages.

Further related work. The investigation of language decompositions goes back to Conway [9], who was interested in expressing a regular event E in the form $f(F_1, F_2, \ldots)$, wherein f is a regular function and F_i are regular events. Language equations form a broad framework in formal language theory in which such kinds of questions are considered (see [15] for a recent overview). The primality question for regular languages [19, 20] is a special case of a language equation, which has been studied in depth, both for finite and infinite languages [19, 10, 2, 13, 18, 20, 23].

In the database theory context, there is a connection with the work of Calvanese et al. [7]. However, their intention is orthogonal to ours. Stated in our framework, they would start from a global schema D and a typing S and ask for a maximal (regular) language of distributed strings w for which S is sound for (D, w).

2. DEFINITIONS AND NOTATION

PROVISO. In this article, all languages are regular. Thus "language" always means "regular language".

2.1 Automata and regular expressions

By Σ we always denote a finite alphabet. A *(nondeterministic, finite) automaton* (or *NFA*) A is a tuple $(Q, \Sigma, \delta, I, F)$, where Q is a set of states, $\delta : Q \times \Sigma \to \mathcal{P}(Q)$ is a transition function, I is the set of initial states and F is the set of accepting states. An automaton is deterministic (or a *DFA*), if I is a singleton and $|\delta(q, a)| \leq 1$, for all $q \in Q, a \in \Sigma$. By δ^* we denote the extension of δ to strings, i.e., $\delta^*(q, w)$ is the set of states that can be reached from q by reading w.

For simplicity we allow in DFA that for some q, a, $\delta(q, a) = \emptyset$. Thus, we do not need non-accepting sink states but rather use undefined transitions to "stop" a run of an automaton. As a consequence, in a minimal DFA, from all states an accepting state is reachable.

An *alternating (finite) automaton* (or *AFA*) A is a tuple $(Q, \Sigma, \delta, I, F)$, which is defined just as in an NFA but where Q is partitioned into E (existential states) and U (universal states). The existential states behave exactly as in an NFA. That is, for an existential state q, if $\delta(q, a) = P$, there exists an accepting run for the remainder of the input word, starting from at least one state in P. The universal states q require that, if $\delta(q, a) = P$, there exists an accepting run for the remainder of the input word, starting from *every* state in P. For details we refer to, e.g., [24].

	LOC	ML	PERF	∃−LOC	∃−ML	∃−PERF
NFA	PSPACE-c	PSPACE-h	PSPACE-c	PSPACE-h in 2-EXPSPACE	PSPACE-h in 2-EXPSPACE	PSPACE-c

Table 1: The complexity results from Abiteboul et al. [1].

	LOC	ML	PERF	∃−2LOC	∃−2ML	∃−PERF
NFA	PSPACE-c [1]	**PSPACE-c** (5.2)	PSPACE-c [1]	PSPACE-h [1] **in NEXPTIME** (6.3)		PSPACE-c [1]
DFA	PSPACE-c [14]	**PSPACE-c** (5.5)	**in PTIME** (3.5)	**PSPACE-c** (6.9)		**in PTIME** (3.5)
DRE	**PSPACE-c** (5.8)	**PSPACE-c** (5.9)	**in PTIME** (3.7)	**PSPACE-h** (6.17)	**PSPACE-h** (6.19) **in EXPTIME**	**in PTIME** (3.7)

Table 2: Summary of complexity results. Results of this paper are written in bold. The result for ML(NFA) was already stated in [1], the current paper presents a corrected proof. All these results also hold for DTDs and XML Schemas using NFAs, DFAs, and DREs as content models. The numbers between brackets indicate the theorem numbers in which the results are proved. We also prove that ∃−LOC(NFA) and ∃−ML(NFA) are in EXPSPACE in general (Theorem 6.2).

The regular expressions over Σ are defined as follows: ε and every Σ-symbol is a regular expression; and whenever r and s are regular expressions, then so are (rs), $(r + s)$, and $(s)^*$. For readability, we usually omit parentheses in examples. The language defined by a regular expression r, denoted by $L(r)$, is defined as usual.

It is well-known that XML schema languages use deterministic (sometimes also called one-unambiguous) regular expressions. Formally, let $\bar{r}$ stand for the regular expression obtained from r by replacing the i-th occurrence of alphabet symbol a in r by a_i, for every i and a. For example, for $r = b^*a(b^*a)^*$ we have $\bar{r} = b_1^*a_1(b_2^*a_2)^*$. A regular expression r is *deterministic* if there are no strings wa_iv and wa_jv' in $L(\bar{r})$ such that $i \neq j$. We denote the class of deterministic regular expressions by DRE. The expression $(a + b)^*a$ is not deterministic as already the first symbol in the string aaa could be matched by either the first or the second a in the expression. The equivalent expression $b^*a(b^*a)^*$, on the other hand, is deterministic. Brüggemann-Klein and Wood showed that not every (nondeterministic) regular expression is equivalent to a deterministic one [6]. Thus, semantically, not every regular language can be defined with a deterministic regular expression. We call a regular language *DRE-definable* if there exists a deterministic regular expression defining it. The canonical example for a language that is not DRE-definable is $(a + b)^*a(a + b)$.

2.2 Typings

Let Σ_f be a set of *function calls*, typically written as f or f_1, f_2, etc. We recall the following notions from Abiteboul et al. [1].

DEFINITION 2.1. A *distributed string* is a string $w = w_0f_1w_1 \dots f_nw_n$, where $n \in \mathbb{N}$, $w_i \in \Sigma^*$ and $f_i \in \Sigma_f$, for each i. We write $w(f_1 \cdots f_n)$ for w if we want to emphasize the function calls. A *design* D is a pair (L, w) consisting of a language L and a distributed string w. We often specify designs as (A, w) or (R, w) for an automaton A or a regular expression R.

DEFINITION 2.2. A typing τ for (L, w) is a sequence $(L_1, \dots, L_n)$ of languages over Σ. We write $w(\tau)$ for the language

$$\{w_0v_1w_1 \cdots v_nw_n \mid v_i \in L_i, 1 \leq i \leq n\}.$$

Given a design $D = (L, w)$ and a typing τ we call τ

- a *sound typing* for D, if $w(\tau) \subseteq L$,
- a *complete typing* for D, if $w(\tau) \supseteq L$,
- a *local typing* for D, if $w(\tau) = L$, i.e., if it is sound and complete,
- a *maximal typing* for D, if it is sound and there exists no sound typing τ' for D, such that $\tau \subsetneq \tau'$, where inclusion is defined componentwise.
- a *perfect typing* for D, if it is local and if for each sound typing τ' for D it holds $\tau' \subseteq \tau$.

2.3 Algorithmic problems

In this paper, we consider the following algorithmic problems. Given a design $D = (L, w)$ and a typing τ,

LOC: check whether τ is a local typing for D;

ML: check whether τ is maximal and local for D;

PERF: check whether τ is a perfect typing for D.

Given a design $D = (L, w)$,

∃−LOC: check whether there exists a local typing for D;

∃−ML: check whether there exists a maximal local typing for D;

∃−PERF: check whether there exists a perfect typing for D.

For a $k \in \mathbb{N}$, we denote by ∃−kLOC (resp., ∃−kML) the problem ∃−LOC (resp., ∃−ML) where w in the given design (L, w) only contains k function calls.

The complexity of these problems might depend on the formalism in which the language L is given and in which the typing has to be specified. For simplicity, we only study cases where these two formalisms coincide. More precisely, we consider NFA (as in [1]), DFA, and DREs as specification formalisms. We denote the resulting algorithmic problems as in LOC(DFA), where L and the target typing are specified by DFA. Since not all regular languages can be defined by DREs, we need to make clear what we mean by ML(DRE).

In ML(DRE) we want to know wheter τ is local and there exists no sound *DRE-definable* typing τ' such that $\tau \subsetneq \tau'$.[5]

Tables 1 and 2 summarize our and previous complexity results for these problems.

2.4 Typings and Regular Languages

We recall some results on language equations that have direct consequences for the typing problem. The next theorem follows immediately from Corollary 13 in [3].

THEOREM 2.3 ([3]). *Let $D = (L, w)$ be a design. If D has a local (even: non-regular) typing then it also has a regular, maximal local typing.*

This theorem holds independently of the formalism in which L is specified, as the considered problems are defined with respect to the languages. It gives a good reason to restrict attention to regular typings as was suggested in [1] and is also done in this paper. One particular consequence of this theorem is that the problems $\exists-$LOC(NFA) and $\exists-$ML(NFA) are identical and that we can therefore merge these entries in Table 2. The same holds for $\exists-$LOC(DFA) and $\exists-$ML(DFA). However, the existence of local typings does not guarantee the existence of local typings specified by DREs (Theorem 6.13) and the existence of local typings specified by DREs does not guarantee the existence of maximal local typings specified by DREs (Theorem 6.15).

3. PERFECT TYPINGS

One of the main results of [1] is that, if a perfect typing exists, there is only *one* candidate typing that needs to be checked and that an NFA can be efficiently constructed (the *perfect automaton* in [1]) from which this typing can be directly inferred. If this typing is local then it is perfect. Therefore, PERF(NFA) can be solved by generating the candidate typing, testing whether it is local, and verifying whether it is equivalent to the typing in the input.

We recall the complexity results from Abiteboul et al. [1]:

THEOREM 3.1 ([1]).
(a) PERF(NFA) *is* PSPACE*-complete, and*
(b) $\exists-$PERF(NFA) *is* PSPACE*-complete.*

The PSPACE-hardness for both problems comes from testing whether the generated candidate typing is local. In other words, these problems are PSPACE-hard because testing language equivalence for NFAs is PSPACE-hard.

This motivated us to study the perfect typing problems for DFAs and for deterministic regular expressions, which are known to have a PTIME language equivalence test.

3.1 Perfect Typings for DFAs

We first study the perfect typing problems for DFAs and prove that PERF(DFA) and $\exists-$PERF(DFA) can be solved in polynomial time. Our overall technique is reminiscent to the one used for proving Theorem 3.1, but the details are rather different. From a given design $D = (A, w)$, where A is a DFA, a candidate automaton (i.e., *perfect automaton*) $\hat{\Omega}(A, w)$ representing a typing τ can be computed in polynomial time such that D has a perfect typing if and only if $w(\tau) = L(A)$. However, two remarks are essential here, in order to understand the new difficulties: (1) the construction of $\hat{\Omega}(A, w)$ is completely different from the construction in [1] and (2) it is not straightforward to check $w(\tau) = L(A)$, because $w(\tau)$ is in general non-deterministic (this non-determinism arises from the choice of remaining in a type τ_i or reading the string w_i to advance to τ_{i+1}). Even if τ consists only of DFAs, the equivalence test $w(\tau) = L$ is PSPACE-complete in general.[6] We therefore need to adopt an approach in which we need more structural insight in the problem, which is exactly our challenge.

Given a design $D = (A, w)$, where A is a DFA $(Q, \Sigma, \delta, s, F)$ and w is a distributed string $w = w_0 f_1 \dots f_n w_n$, we construct the candidate automaton $\Omega(D)$ as follows. We use the extended alphabet $\hat{\Sigma} = \Sigma \uplus \{\sigma_0, \dots, \sigma_n\}$ and the homomorphism $h : \hat{\Sigma}^* \to \Sigma^*$, where $h(a) = a$ for any $a \in \Sigma$ and $h(\sigma_i) = w_i$ for any $i \in \{1, \dots, n\}$.

By $\hat{A}$ we denote the automaton derived from A by applying the inverse homorphism h^{-1} to A. More precisely, $\hat{A} = (Q, \hat{\Sigma}, \hat{\delta}, s, F)$, where

$$\hat{\delta} = \delta \cup \{(q_a, \sigma_i, q_b) | q_b \in \delta^*(q_a, w_i)\}.$$

Since Σ and $\{\sigma_1, \dots, \sigma_n\}$ are disjoint, $\hat{A}$ is deterministic. Furthermore it can be constructed in polynomial time.

The *perfect automaton* $\hat{\Omega} = \hat{\Omega}(A, w)$ is defined as the minimal DFA for $L(\hat{A}) \cap L(\sigma_0 \Sigma^* \sigma_1 \Sigma^* \dots \Sigma^* \sigma_n)$. We can construct $\hat{\Omega}$ in polynomial time by performing the standard product construction on $\hat{A}$ and the (trivial) linear size deterministic automaton for $\sigma_0 \Sigma^* \sigma_1 \Sigma^* \dots \Sigma^* \sigma_n$. Recall our convention that minimal DFA do not have (rejecting) sink states and therefore, for some q and σ_i, $\hat{\delta}(q, \sigma_i)$ might be empty. It should be noted that, as $\hat{\Omega}$ is minimal, $\hat{\Omega}$ only depends on D, not on A.

EXAMPLE 3.2. Figure 3 illustrates our construction with two designs. The DFA A_1 of the design $D_1 = (A_1, f_1 f_2)$ is shown in Figure 3(a) (without the dashed transitions). $\hat{A}_1$ results from adding the dashed self-loops, as the strings w_0, w_1 and w_2 are empty. The perfect automaton $\hat{\Omega}(D_1)$ is shown in Figure 3(b). Later, we will see that this design does not have a perfect typing.

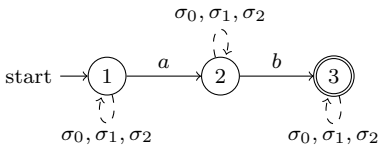
The lower part of Figure 3 gives a more complicated example, where a perfect typing actually exists. The design is $D_2 = (A_2, f_1 b c f_2)$, where A_2 is the DFA of Figure 3(d), without the dashed transitions. The DFA $\hat{A}(D_2)$ is the automaton in Figure 3(d) with the dashed transitions. The two self-loops labeled with σ_0 and σ_2 at each state result again from the empty strings w_0 and w_2. The perfect automaton is shown in Figure 3(e). ■

For each $1 \le i \le n$, we define the *local automaton* Ω_i as follows. First, let $\hat{\Omega}_i$ be the automaton obtained from $\hat{\Omega}$ by choosing *(i)* as initial states those states q with some transition (r, σ_{i-1}, q), and *(ii)* as final states those states p with some transition (p, σ_i, r'). Then, Ω_i is the automaton obtained from $\hat{\Omega}_i$ by removing all transitions labeled with some σ_j.

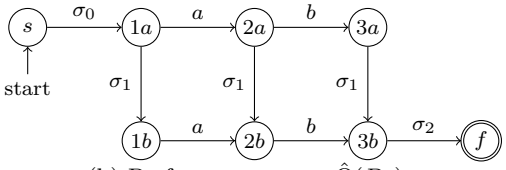
Notice that, since A is deterministic, the only nondeterminism of Ω_i is the freedom to choose an initial state. We

[5] One could define this problem in two different manners: either τ' can be regular, or needs to be DRE-definable. From our proof it follows that these two problems coincide.

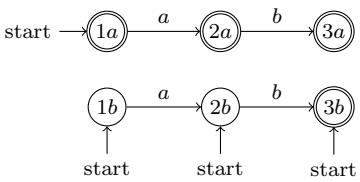
[6] Already deciding whether $L(A) = L(A_1) \cdot L(A_2)$ for DFAs A, A_1, and A_2 is PSPACE-complete in general, see [14] and Theorem 5.7.



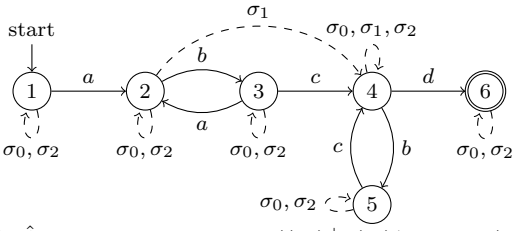
(a) DFA $\hat{A}_1$ for the design $D_1 = (\{ab\}, f_1 f_2)$.



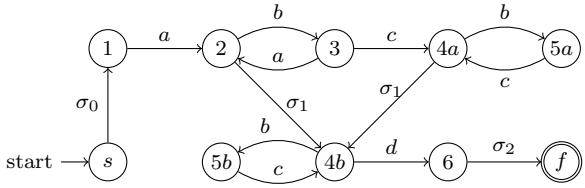
(b) Perfect automaton $\hat{\Omega}(D_1)$.



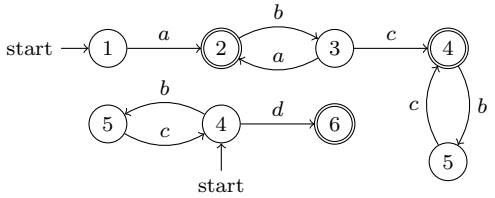
(c) Local candidate automata $\Omega_1(D_1)$ and $\Omega_2(D_1)$.



(d) DFA $\hat{A}_2$ for the design $D_2 = ((ab)^+c(bc)^*d, f_1bcf_2)$.



(e) Perfect automaton $\hat{\Omega}(D_2)$.



(f) Local candidate automata $\Omega_1(D_2)$ and $\Omega_2(D_2)$.

Figure 3: Two designs: (a)–(c) has no perfect typing, (d)–(f) has a perfect typing.

write $\vec{\Omega}$ for $(\Omega_1, \ldots, \Omega_n)$ and τ_Ω for the typing $(L(\Omega_1), \ldots, L(\Omega_n))$.

Figures 3(c) and 3(f) display the respective local automata for the designs of Example 3.2.

We need the following technical lemma:

LEMMA 3.3. *Let $w = w_0 f_1 \ldots f_n w_n$ be a distributed string, A a DFA and $\tau = (L_1, \ldots, L_n)$ a sound typing for (A, w). Let τ_Ω be the typing obtained from $\hat{\Omega}(A, w)$ as described above. Then $\tau \subseteq \tau_\Omega$.*

The following theorem is the technical core of this section. It proves that to test whether a design has a perfect typing it suffices to test whether all local candidate automata have one initial state. Furthermore, the perfect typing is simply the vector of local candidate automata.

THEOREM 3.4. *Let $w = w_0 f_1 \ldots f_n w_n$ be a distributed string and A a DFA, such that $L(A) \subseteq w_0 \Sigma^* w_1 \cdots \Sigma^* w_n$. Let $\hat{\Omega}$, τ_Ω, and $\vec{\Omega} = (\Omega_1, \ldots, \Omega_n)$ be defined as above. Then the following statements are equivalent.*

(a) There is a perfect typing for (A, w).

(b) τ_Ω is a perfect typing for (A, w).

(c) τ_Ω is a sound typing for (A, w).

(d) For each i, Ω_i has exactly one initial state.

PROOF. We show the implications (a) $\Rightarrow$ (d) $\Rightarrow$ (c) $\Rightarrow$ (b) $\Rightarrow$ (a).

(a) $\Rightarrow$ (d): Let $\tau = (L_1, \ldots, L_n)$ be a perfect typing for (A, w). Towards a contradiction, assume that, for some $1 \leq i \leq n$, $p \neq q$ are initial states of Ω_i. Since by definition $\hat{\Omega}$ is minimal, there exists a string $u = u_i \sigma_i \cdots u_n \sigma_n$ such that $\delta^*(p, u) \in F$ and $\delta^*(q, u) \notin F$ or vice versa. We assume w.l.o.g. that $\delta^*(p, u) \in F$ — the other case is symmetric. Since by minimality of $\hat{\Omega}$ every state occurs in some accepting run and $L(\hat{\Omega}) \subseteq \sigma_0 \Sigma^* \sigma_1 \cdots \sigma_{n-1} \Sigma^* \sigma_n$, there exist strings

- $v = v_i \sigma_i \cdots v_n \sigma_n$ with $\delta^*(q, v) \in F$,
- $u' = \sigma_0 u_1 \sigma_1 \cdots u_{i-1} \sigma_{i-1}$ with $\delta^*(s, u') = p$,
- $v' = \sigma_0 v_1 \sigma_1 \cdots v_{i-1} \sigma_{i-1}$ with $\delta^*(s, v') = q$.

Thus, $u'u$ and $v'v$ are both accepted by $\hat{\Omega}$ and therefore $(\{u_1\}, \ldots, \{u_n\})$ and $(\{v_1\}, \ldots, \{v_n\})$ are sound typings for (A, w). By perfectness of τ, both these typings are included in τ, hence, for each i, $\{u_i, v_i\} \subseteq L_i$. But this yields a contradiction as $u'v$ is not accepted by $\hat{\Omega}$, thus τ is not sound. Thus, we can conclude that (d) holds.

(d) $\Rightarrow$ (c): Let, for each i, q_i be the unique[7] initial state of Ω_i. Let, for each i, $v_i \in L(\Omega_i)$. We need to show that $w_0 v_1 w_1 \cdots v_n w_n \in L(A)$.

For each i, let $s_i = \hat{\delta}^*(q_i, v_i)$. By construction of $\hat{\Omega}_i$ and uniqueness of initial states, we have that $\hat{\delta}^*(s_{i-1}, \sigma_{i-1}) = q_i$ for each i (where s_0 is interpreted as the initial state of $\hat{\Omega}$). Furthermore, $q_a = \hat{\delta}(s_n, \sigma_n)$ is the unique accepting state of $\hat{\Omega}$. Together,

$$s_0 \xrightarrow{\sigma_0} q_1 \xrightarrow{v_1}{}^* s_1 \xrightarrow{\sigma_1} \cdots \xrightarrow{\sigma_{n-1}} q_n \xrightarrow{v_n}{}^* s_n \xrightarrow{\sigma_n} q_a$$

[7] It should be noted that by construction q_i is also the unique initial state of $\hat{\Omega}_i$.

is an accepting computation of $\hat{\Omega}$, hence for $\hat{A}$ and therefore

$$s_0 \xrightarrow{w_0}{}^* q_1 \xrightarrow{v_1}{}^* s_1 \xrightarrow{w_1}{}^* \cdots \xrightarrow{w_{n-1}}{}^* q_n \xrightarrow{v_n}{}^* s_n \xrightarrow{w_n}{}^* q_a$$

is an accepting computation of A.
(c) $\Rightarrow$ (b): Let $v = w_0v_1w_1 \cdots v_nw_n \in L(A)$. It follows that $\tau = (\{v_1\}, \ldots, \{v_n\})$ is a sound typing for (A, w). By Lemma 3.3, $\tau \subseteq \tau_\Omega$ and thus $v \in w(\tau_\Omega)$, therefore τ_Ω is complete, hence local. Applying Lemma 3.3 again, immediately yields that τ_Ω is perfect.
(b) $\Rightarrow$ (a): Immediate. □

Using Theorem 3.4, we can prove that the perfect typing problems are tractable.

THEOREM 3.5. *(a)* PERF(DFA) *is in* PTIME *and (b)* $\exists-$PERF(DFA) *is in* PTIME.

3.2 Deterministic Regular Expressions

In real DTDs and XML Schema specifications content models are described by *deterministic* regular expressions ([5, 12]). This raises the question how to solve the perfect typing problem for DREs. We first show that the case of deterministic regular expressions is quite different from the case of finite automata. In particular, there are designs with perfect typings that cannot be specified by DREs.

THEOREM 3.6. *There is a design $D = (R, w)$ with a DRE R for which the (unique) perfect typing is not expressable by DREs.*

PROOF. We show that the perfect typing for the design D_2 of Example 3.2 cannot be specified by DREs. The global schema of D_2 is specified by the DRE $R = (ab)^+c(bc)^*d$. As we argued in Example 3.2, the DFAs $\Omega_1 = \Omega_1(D_2)$ and $\Omega_2 = \Omega_2(D_2)$ describe a perfect typing $\tau = (L(\Omega_1), L(\Omega_2))$ for D_2, since they both have only one initial state. By using techniques from [6] it can be shown that there can be no DRE for $L(\Omega_1)$. □

Theorem 3.6 shows that DREs require some care. However, perfect typings are still feasible as stated in the next result.

THEOREM 3.7. *(a)* PERF(DRE) *is in* PTIME *and (b)* $\exists-$PERF(DRE) *is in* PTIME.

4. NORMAL FORM TYPINGS

For a given design there can be an infinite number of local typings. We show in this section that we can reduce the search space considerably by only considering typings of particular normal forms.

Let, in the following, $A = (Q, \Sigma, \delta, I, F)$ always denote some NFA.

An *A-transformation* is a mapping $Q \to \mathcal{P}(Q)$, i.e., a function that maps states of A to sets of states of A. For a string w, we denote by T^A_w the A-transformation *induced by* w, i.e., the function $p \mapsto \delta^*(p, w)$. Given an A-transformation T we write $L_{\text{trans}}(A, T)$ for the set of strings w with $T^A_w = T$. We say a typing $(L_1, \ldots, L_n)$ is in *A normal form (A-NF)* if each L_i is a union of languages of the form $L_{\text{trans}}(A, T)$.

If A is a DFA we consider a stronger normal form: For two sets X, Y of states of A let $L_\cap(A, X, Y)$ denote the set of all strings w, for which $\delta^*(p, w) \in Y$, for every $p \in X$. A typing $(L_1, \ldots, L_n)$ is in *strong A-normal form (strong A-NF)*, if

(1) each L_i is of the form $L_\cap(A, X_i, Y_i)$, for some $X_i, Y_i \subseteq Q$,

(2) X_1 is a singleton, and

(3) $\delta^*(p, w_n) \subseteq F$, for every $p \in Y_n$.

The idea behind the strong A-NF is, that each set X is chosen as a subset of the initial states of the corresponding local automaton as constructed in section 3. Each set Y is then chosen as the set of states, such that a state of the next X-set is reached by reading the next string w_i.

REMARK 4.1. The specialisation of the strong A-NF with $i = 2$ and all $w_i = \varepsilon$ was already used by Salomaa et al. [19] under the term *decomposition sets*.

REMARK 4.2. It follows from the results of Section 3 that a perfect typing for a design (A, w) with a DFA A is always in strong A-NF.

Even though local or maximal local typings do not need to be of this particular simple type (as we will see below), we show next that A-NF typings and strongly A-NF typings deserve their names.

THEOREM 4.3. *Let A be an NFA, and $\tau = (L_1, \ldots, L_n)$ a local typing for the design $D = (A, w(f_1 \cdots f_n))$.*

(a) Then there exists an A-NF local typing τ' for D such that $\tau \subseteq \tau'$.

(b) If A is a DFA there exists a strong A-NF local typing τ' for D such that $\tau \subseteq \tau'$.

PROOF. (a): For each $i = 1, \ldots, n$ we let

$$L'_i = \bigcup_{w \in L_i} L_{\text{trans}}(A, T^A_w),$$

i.e., the set of strings for which there is some $w \in L_i$ with the same A-transition. Let $\tau' = (L'_1, \ldots, L'_n)$. As, in particular, each string $w \in L_i$ is in L'_i, we immediately get $\tau \subseteq \tau'$.

It remains to show that τ' is a sound typing for (A, w). To this end, let, for each i, $v_i \in L'_i$. For each i, there is some $u_i \in L_i$ with $T^A_{u_i} = T^A_{v_i}$. As τ is a sound typing, A has an accepting run on $w_0u_1w_1 \cdots u_nw_n$. As each v_i has the same A-transformation as the respective u_i, $w_0v_1w_1 \cdots v_nw_n$ is accepted by A as well.

(b): Let s denote the initial state of A. For each $i = 1, \ldots, n$, we let $L'_i = L_\cap(A, X_i, Y_i)$ where $X_1 = \delta^*(s, w_0)$, $X_i = \bigcup_{q \in Y_{i-1}} \delta^*(q, w_{i-1})$, for $i \in \{2, \ldots, n\}$, and $Y_i = \bigcup_{q \in X_i} \bigcup_{w \in L_i} \delta^*(q, w)$, for $i \in \{1, \ldots, n\}$.

Let $\tau' = (L'_1, \ldots, L'_n)$. As, in particular, each string $w \in L_i$ is also in L'_i, we immediately get $\tau \subseteq \tau'$.

Clearly, τ' fulfills conditions (1) and (2) of the strong normal form. It remains to show that it also fulfills condition (3) and that it is a sound typing for (A, w). We use the following claim, which we prove later.

CLAIM 4.4. *For each $i \in \{1, \ldots, n\}$ the following conditions hold.*

(a) For each $q \in Y_i$, there is a string $v \in w_0L_1 \cdots w_{i-1}L_i$ such that $\delta^*(s, v) = \{q\}$.

(b) $w_0L'_1w_1 \cdots w_{i-1}L'_i \subseteq L(A^{Y_i})$.

Here, A^{Y_i} denotes the automaton A with final state set Y_i, i.e., $A^{Y_i} := (Q, \Sigma, \delta, I, Y_i)$.

Towards (3) let $p \in Y_n$. By (a) there is a string $v \in w_0 L_1 w_1 \cdots L_n$ such that $\delta^*(s, v) = \{p\}$. As τ is sound, $vw_n \in L(A)$ and thus $\delta^*(p, w_n) \subseteq F$.

It remains to show the soundness of τ'. By (b) for every string $vw_n \in w_0 L'_1 w_1 \cdots w_{n-1} L'_n w_n$ we have $\delta^*(s, v) \subseteq Y_n$ and thus, by (3) $\delta^*(s, vw_n) \subseteq F$, yielding soundness of τ'. □

REMARK 4.5. Clearly, if τ is a local (maximal local, perfect) typing then τ' is local (maximal local, perfect) as well. Therefore, even if not every typing *has* an equivalent normal form typing but only *is contained* in a sound normal form typing, we consider the term "normal form" adequate.

We still need to proof Claim 4.4.

PROOF. We let $Y_0 = \{s\}$ and prove (a) and (b) by simultaneous induction on i, for every $i \in \{0, \ldots, n\}$. Clearly (a) and (b) hold for $i = 0$ (as they only refer to the empty string).

Now let $i \geq 1$ and $q \in Y_i$. By definition of Y_i there are $p \in X_i$ and $w \in L_i$ such that $\delta^*(p, w) = \{q\}$. By definition of X_i, $\delta^*(r, w_{i-1}) = \{p\}$, for some $r \in Y_{i-1}$. By induction, there is a string $v \in w_0 L_1 \cdots w_{i-1} L_{i-1}$ such that $\delta^*(s, v) = \{r\}$. Thus, $\delta^*(s, vw_{i-1}w) = \{q\}$ and (a) follows.

Now let $w_0 v_1 \cdots w_{i-1} v_i$ be a string in $w_0 L'_1 \cdots w_{i-1} L'_i$. Let p be such that $\delta^*(s, w_0 v_1 \cdots v_{i-1}) = \{p\}$. By induction, $p \in Y_{i-1}$. By definition of L'_i there is a state $q \in Y_i$ such that $\delta^*(p, w_{i-1} v_i) = \{q\}$. Thus, $\delta^*(s, w_0 v_1 \cdots w_{i-1} v_i) = \{q\} \subseteq Y_i$ and (b) follows. □

Theorem 4.3 shows that, if one is interested in the existence of a (local, maximal local, perfect) typing, it is always sufficient to look for (strong) A-NF typings. Furthermore, it shows that *every* maximal local typing is in normal form.

The next theorem shows why normal forms are interesting from a complexity-theoretic point of view: we can define the languages in normal form typings by means of "small" finite automata.

THEOREM 4.6. *Let A be an NFA, $D = (A, w(f_1 \cdots f_n))$ a design, and $\tau = (L_1, \ldots, L_n)$ a typing for D.*

(a) If τ is in A-normal form then, for each i, $1 \leq i \leq n$, there is a DFA B of exponential size in $|A|$ such that $L(B) = L_i$.

(b) If A is a DFA and τ is in strong A-normal form, then, for each i, $1 \leq i \leq n$, there is an AFA B of polynomial size in $|A|$ such that $L(B) = L_i$.

PROOF. (a) The DFA B simply keeps track of the transformation T_w^A induced by the input string.

Let $A = (Q, \Sigma, \delta, I, F)$ be an NFA with state set $Q = \{q_1, \ldots, q_m\}$ and let $L = \bigcup_{j=1}^{k} L_{\text{trans}}(A, T_j)$ be an A-NF language. The DFA B is defined as $(\mathcal{P}(Q)^m, \Sigma, \delta_B, I_B, F_B)$, where

- the transition function δ_B is defined by

$$\delta_B((Q_1, \ldots, Q_m), a) = (\delta(Q_1, a), \ldots, \delta(Q_{|Q|}, a)).$$

  Here, as usual $\delta_A(Q_i, a) = \bigcup_{p \in Q_i} \delta_A(p, a)$, for every i;

- the initial state set I_B is $(\{q_1\}, \ldots, \{q_m\})$;

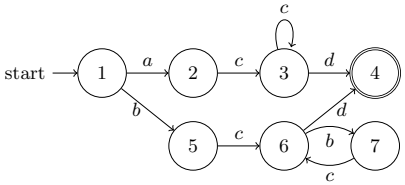


Figure 4: Automaton A of Example 4.7

- F_B consists of all states $(T_j(q_1), \ldots, T_j(q_m))$, $j \leq k$.

(b) Let A be a DFA and $\tau = (L_1, \ldots, L_n)$ a typing for D in strong A-normal form. For each i, $L_i = L_\cap(A, X, Y)$ for some X and Y. Let B be the AFA that first universally branches to the states in X and then simulates deterministically (on all branches) A on w. Its accepting states are the states in Y. □

We note that the bounds of Theorem 4.6 do *not* apply to DFAs for the languages $w(\tau)$ as the concatenation of languages re-introduces nondeterminism. However, we can conclude a double-exponential size bound for DFAs for the languages $w(\tau)$.

It is tempting to hope for stronger normal forms for typings. For example, if in Theorem 4.3(b) all languages in τ' were of the form $L_\cap(A, X, Y)$ with singleton X, then we could use polynomial-size NFAs instead of polynomial-size AFAs in Theorem 4.6(b). However, the following example shows that this is not possible and that therefore, our normal forms are, in a sense, optimal.

EXAMPLE 4.7. Let A be a DFA for the language $ac^+d + (bc)^+d$. and $D = (A, f_1 c f_2)$. A is depicted in Figure 4. The only local (and thus also maximal local) typing for D is $\tau = (ac^* + b(cb)^*, d) = (L_\cap(A, \{1\}, \{2, 3, 5, 7\}), L_\cap(A, \{3, 6\}, \{4\}))$.

Notice that there is no single state s of A such that there exists a local typing of the form $(L_\cap(A, X_1, Y_1), L_\cap(A, \{s\}, Y_2))$ for D. ■

5. VERIFICATION OF TYPINGS

In this section, we study the complexity of testing whether a given typing is local or maximal local for a given design.

5.1 Non-deterministic Content Models

We first consider designs where the schema is specified by an NFA and recall the known results. Abiteboul, Gottlob, and Manna [1] proved the following:

THEOREM 5.1 ([1]). LOC(NFA) *is* PSPACE*-complete.*

They furthermore stated the following result:

THEOREM 5.2. ML(NFA) *is* PSPACE*-complete.*

However, the proof in [1] is not correct. It claims that, for a design (L, w) a typing $\tau = (L_1, \ldots, L_n)$ is *not* maximal, if there is an i, such that

$$w(L_1, \ldots, L_{i-1}, \overline{L_i}, L_{i+1}, \ldots, L_n) \cap L \neq \emptyset.$$

This is not true: if $L = \{a, aa\}$ and $w = f_1 f_2$, the typing $\tau = (L_1, L_2)$ with $L_1 = \{\varepsilon, a\}$ and $L_2 = \{a\}$ is maximal, even though the string aa is in $w(L_1, \overline{L_2}) \cap L$. Nevertheless, as we prove here, the result itself is correct.

PROOF. Let $D = (A, w)$. We first show that a local typing $\tau = (L_1, \ldots, L_n)$ is *not* maximal for D if and only if there is an i, $1 \leq i \leq n$ and an A-transformation T such that

(1) $(L_1, \ldots, L_{i-1}, L_{\text{trans}}(A, T), L_{i+1}, \ldots, L_n)$ is sound for D and

(2) $L_{\text{trans}}(A, T) - L_i \neq \emptyset$.

The "if" statement holds by definition of "maximal". For the "only if" statement let us assume that $\tau \subsetneq \tau''$, for some local typing τ''. By Theorem 4.3, there is an A-NF typing $\tau' = (L'_1, \ldots, L'_n)$ such that $\tau'' \subseteq \tau'$, thus $\tau \subsetneq \tau'$. Therefore, there is some i such that $L_i \subsetneq L'_i$. By definition of A-NF typings there is an A-transformation T such that $L_{\text{trans}}(A, T) \subseteq L'_i$ but $L_{\text{trans}}(A, T) \not\subseteq L_i$.

Whether a given typing τ is maximal and local can thus be tested as follows.

(a) Test whether τ is local.

(b) Guess i and T.

(c) Check (1) and (2) above.

To test (1) it is sufficient to construct an NFA A' for

$$w(L_1, \ldots, L_{i-1}, L_{\text{trans}}(A, T), L_{i+1}, \ldots, L_n) \cap \overline{L}$$

and to verify that $L(A') = \emptyset$. It is not hard to see, that there is such an NFA of exponential size which can be represented succinctly in polynomial space and therefore its non-emptiness can be tested in (nondeterministic thus also deterministic) polynomial space.

Condition (2) can be easily tested in polynomial space. □

5.2 Deterministic Content Models

The PSPACE-hardness of LOC(NFA) immediately follows from the PSPACE-hardness of language equivalence for NFAs [1]. In the light of section 3, one could hope for a significant drop of complexity when moving to deterministic content models. However, quite contrary to Section 3, this is neither the case for DFA nor for DREs.

5.2.1 Deterministic Finite Automata

For proving the PSPACE-hardness of LOC(DFA), we recall a closely related problem from Formal Language Theory: the ConcatenationUniversality problem asks for given DFAs A_1, A_2 whether $L(A_1) \cdot L(A_2) = \Sigma^*$.

THEOREM 5.3 (JIANG AND RAVIKUMAR [14]). ConcatenationUniversality *is* PSPACE*-complete.*

Actually, this result is not explicitly stated but is implicit in the proof of Theorem 4.1 in [14].

It is easy to see that ConcatenationUniversality for DFAs A_1 and A_2 is just a special case of LOC(DFA): just take $D = (\Sigma^*, f_1 f_2)$ and $\tau = (L(A_1), L(A_2))$. Furthermore, since the upper bound of Theorem 5.1 carries over to DFAs, we immediately get the following corollary.

COROLLARY 5.4. LOC(DFA) *is* PSPACE*-complete. Furthermore, it is already* PSPACE*-hard for the design* $(\Sigma^*, f_1 f_2)$.

For testing maximal locality, the upper bound immediately follows from Theorem 5.2. We defer the lower bound proof to Lemma 6.8.

THEOREM 5.5. ML(DFA) *is* PSPACE*-complete.*

5.2.2 Deterministic Regular Expressions

From Section 3 we know that typings expressed by DREs require some special care. However, the following result carries over easily from DFAs to DREs since deterministic regular expressions can be translated into DFAs in quadratic time.

LEMMA 5.6. LOC(DRE) *is in* PSPACE.

The corresponding lower bound and the respective results for maximal typings require more work. In particular, we need to revisit the PSPACE lower bound for locality testing for DREs. We will follow a similar outline as in Section 5.2.1 to prove that LOC(DRE) is PSPACE-hard. To this end, we first define ConcatenationUniversality *for DREs*, i.e., the problem to decide, given two DREs R_1 and R_2, whether $L(R_1) \cdot L(R_2) = \Sigma^*$, and prove that it is PSPACE-hard. However, we cannot simply adapt the proof of Theorem 5.3 by Jiang and Ravikumar [14] since that proof does not use DRE-definable languages.

THEOREM 5.7. ConcatenationUniversality *for DREs is* PSPACE*-complete.*

The upper bound is easily obtained by transforming R_1 and R_2 into NFAs and applying Theorem 5.1. The lower bound can be proven by a reduction from the complement of PSPACE-complete CorridorTiling [21].

COROLLARY 5.8. LOC(DRE) *is* PSPACE*-complete. Furthermore, it is already* PSPACE*-hard for the design* $(\Sigma^*, f_1 f_2)$.

For testing maximal locality, one option could be to translate the given DREs into NFAs and to use the upper bound algorithm from Theorem 5.2. It is not obvious that this is correct: a typing defined by DREs could be found non-maximal by the algorithm of Theorem 5.2 because there is a larger typing that is not DRE-definable. However, if there exists a larger typing that is not DRE-definable, then there is also a larger typing that is DRE-definable. The reason is that, for every DRE-definable language L and string w, the language $L \cup \{w\}$ is also DRE-definable (Lemma 10 in [4]). This shows that ML(DRE) is in PSPACE. We defer the lower bound proof to Section 6 (Lemma 6.18), since it heavily builds on a proof presented there.

THEOREM 5.9. ML(DRE) *is* PSPACE*-complete.*

6. EXISTENCE OF TYPINGS

As usual, we first recall the complexity results of [1].

THEOREM 6.1 ([1]). *(a)* $\exists-$LOC(NFA) *is* PSPACE*-hard and (b)* $\exists-$LOC(NFA) *is in* 2-EXPSPACE.

6.1 Non-deterministic Content Models

For the general problem, Theorems 4.3 and 4.6 allow us to improve the 2-EXPSPACE upper bounds of [1] to EXPSPACE:

THEOREM 6.2. $\exists-$LOC(NFA) *is in* EXPSPACE.

PROOF. According to Theorem 4.3, it suffices to test A-normal form typings and according to Theorem 4.6, each language L_i in an A-NF typing $(L_1, \ldots, L_n)$ can be represented by a DFA of exponential size in $|A|$. We can conclude that $w_0 L_1 w_1 \cdots L_n w_n$ can be represented by an NFA of exponential size. Since equivalence between such an NFA and A can also be tested in exponential space, we can test $\exists-$LOC(NFA) in EXPSPACE by testing whether any A-NF typing is local for $L(A)$. □

If the distributed string has only two function calls, we can do better.

THEOREM 6.3. $\exists-$2LOC(NFA) *is in* NEXPTIME.

6.2 Deterministic Content Models

In a similar way as LOC(DFA) is related to the problem ConcatenationUniversality, $\exists-$LOC(DFA) is also related to a classical problem from Formal Language Theory. This problem is called the Primality problem. The complexity of Primality has been considered an open problem in Formal Language Theory since the late 90's (see Problem 2.1 in [20], see also [19, 10, 2, 13, 18, 23]). Primality is decidable but no further lower or upper bounds are known [20]. We pinpoint the precise complexity of Primality in Theorem 6.4 below: it is PSPACE-complete.

That the complexity of Primality was open for a long time indicates that it might be non-trivial to figure out the precise complexity of $\exists-$LOC(DFA) and $\exists-$LOC(NFA), as they are in a sense generalizations of Primality. As a step towards an answer to these complexity questions we determine the precise complexity of $\exists-$LOC(DFA) for distributed strings with at most two function calls, a case that already generalizes Primality.

6.2.1 The Language Primality Problem

The Primality problem for formal languages is defined as follows. A *non-trivial decomposition of a language* L is a pair (L_1, L_2) of languages, $L_1 \neq \{\varepsilon\} \neq L_2$ such that $L = L_1 \cdot L_2$. A language is called *prime* if it does *not* have a non-trivial decomposition. Primality asks, given a DFA A, whether $L(A)$ is prime.

THEOREM 6.4. Primality *is* PSPACE*-complete.*

PROOF. We first prove that Primality is PSPACE-hard. We use a polynomial time reduction from the complement of ConcatenationUniversality. Given two DFAs A_1 and A_2, we construct a DFA A, such that $L(A)$ is prime, if and only if $L(A_1) \cdot L(A_2) \neq \Sigma^*$.

To this end, let A_1 and A_2 be two arbitrary DFAs. Without loss of generality, we can assume that $L(A_1)$ and $L(A_2)$ are strict supersets of $\{\varepsilon\}$. Let $\Sigma' = \{a' \mid a \in \Sigma\}$ be a disjoint copy of Σ. Let \$ be a symbol not occurring in Σ or Σ'. By A_1' and A_2' we denote the DFAs resulting from A_1 and A_2 by replacing each character a from Σ with the corresponding character a' from Σ'. We denote the languages of A_1, A_2, A_1' and A_2' by L_1, L_2, L_1' and L_2', respectively. We let A be an automaton for

$$L =_{\text{def}} \Sigma^* \cup L_1\$L_2' \cup L_1'\$L_2 \cup L_1'\$\$L_2'.$$

CLAIM 6.5. *Either there is no nontrivial decomposition of* L *or the only nontrivial decomposition is* (L_a, L_b) *with* $L_a = L_1 \cup L_1'\$$ *and* $L_b = L_2 \cup \$L_2'$.

We now prove that L is not prime, if and only if $L_1 \cdot L_2 = \Sigma^*$. If L is not prime, according to Claim 6.5, the only nontrivial decomposition is (L_a, L_b). Since $L_a \cap \Sigma^* = L_1$, $L_b \cap \Sigma^* = L_2$ and $L \cap \Sigma^* = \Sigma^*$, we can conclude that $L_1 \cdot L_2 = \Sigma^*$.

For the other direction we claim, that if $L_1 L_2 = \Sigma^*$, then (L_a, L_b) is a decomposition of L. Indeed, since each string in L can be written as a concatenation of a string in L_a and a string in L_b and conversely, we have that $L = L_a L_b$.

For the upper bound, we refer to Corollary 6.10. □

6.2.2 Primality versus $\exists-$2LOC(DFA)

We clarify the relation between Primality and $\exists-$2LOC(DFA). Intuitively, one might assume that Primality can be logspace reduced to $\exists-$2LOC(DFA) by simply mapping the DFA A (the input to Primality) to the design $(A, f_1 f_2)$. However, a local typing for this design could yield the trivial decompositions (L_1, L_2) where $L_1 = \{\varepsilon\}$ and $L_2 = L(A)$ or vice versa. Therefore, we reduce from StrongPrimality, the variant of Primality which asks, given a DFA A whether there exists a non-trivial *strong* decomposition (L_1, L_2) of $L(A)$. i.e., where $\varepsilon \notin L_1$ and $\varepsilon \notin L_2$.

THEOREM 6.6. StrongPrimality *is* PSPACE*-complete.*

The proof follows the lines of the proof of Theorem 6.4 .

LEMMA 6.7. $\exists-$LOC(DFA) *is* PSPACE*-hard, already for designs of the form* $(R, f_1 a f_2)$.

PROOF. We reduce from the complement of StrongPrimality. Let A be a DFA for a language L. We can assume w.l.o.g. that L does not contain strings of length at most one.

Let $L^{\#}$ be the language $\{a_1\#a_2\#\ldots\#a_n \mid a_1a_2\ldots a_n \in L\}$ and w the distributed string $f_1\#f_2$, where $\#$ is a symbol not occurring in Σ. It is straightforward to construct a DFA for $L^{\#}$ in logarithmic space. It can be shown that L is strongly prime if and only if there does not exist a local typing for $(L^{\#}, w)$. □

Inspired by the proof of Lemma 6.7 we can now show the following result.

LEMMA 6.8. ML(DFA) *is* PSPACE*-hard.*

6.2.3 Upper Bounds for $\exists-$LOC(DFA)

THEOREM 6.9. $\exists-$2LOC(DFA) *is* PSPACE*-complete.*

PROOF. The lower bound is immediate from Lemma 6.7.

For the upper bound, let $D = (A, w_0 f_1 w_1 f_2 w_2)$ be a design, where A is a DFA. Thanks to Theorem 4.3 it is sufficient to consider typings $\tau = (L_1, L_2)$ in strong normal form, i.e., where $L_i = L_\cap(A, X_i, Y_i)$, for $i \in \{1, 2\}$. Thus, the algorithm guesses such a typing and verifies that it is a local typing for D. As NPSPACE = PSPACE, it only remains to show that the latter can be done in polynomial space.

To this end, let B be the following alternating automaton

(1) It checks that its input is of the form $w_0 u$,

(2) it simulates A on u' and, whenever it enters a state from Y_1 it non-deterministically decides to continue the simulation or to proceed with (3),

(3) it tests that the rest of the string is of the form $w_1 u'$, and

(4) it verifies that $u' \in L_\cap(A, X_2, Y_2) \cdot w_2$ by universally branching to all states $p \in X_i$ and testing that $u' = u''w_2$ with $\delta^*(p, u'') \subseteq Y_2$.

The equivalence of the AFA B with A can be tested in polynomial space (cf. [22]). □

COROLLARY 6.10. Primality *and* StrongPrimality *are in* PSPACE.

PROOF. Both problems can be solved by slightly adapted versions of the algorithm in Theorem 6.9. For Primality, one only needs to check that $L_1 \neq \{\varepsilon\} \neq L_2$. It is easy to see that if there is a non-trivial factorization at all, it can be chosen in strong normal form.

For StrongPrimality one checks whether $(L_1 \setminus \{\varepsilon\}, L_2 \setminus \{\varepsilon\})$ is a local typing. Here, the strong normal form might add ε to (one or both) factors, therefore all strong normal form typings have to be considered. We show that this approach is correct. Suppose that (L_1', L_2') is a decomposition of L such that $\varepsilon \notin L_1'$ and $\varepsilon \notin L_2'$, and suppose that L_1 and L_2 are the respective languages of the strong normal form. By construction, we have that $L_1' \subseteq L_1$ and $L_2' \subseteq L_2$ and by Theorem 4.3 we have that $L_1 L_2 = L$. However, since already $L_1' L_2' = L$ we also have that $(L_1 \setminus \{\varepsilon\})(L_2 \setminus \{\varepsilon\}) = L$. □

The exact complexity of the general ∃−LOC(DFA) problem remains open. Theorem 6.2 immediately yields the following upper bound.

THEOREM 6.11. ∃−LOC(DFA) *is in* EXPSPACE.

At this point, there is no matching lower bound. However, a problem which appears to be very close to ∃−LOC(DFA) is indeed EXPSPACE-complete.

THEOREM 6.12. *Given DFAs A and B and a typing $\tau = (L_1, L_2, L_3)$ in strong A-normal form, it is* EXPSPACE*-complete to decide, whether τ is complete for $(B, f_1 f_2 f_3)$.*

PROOF. The upper bound is obvious. The lower bound is by a reduction from ExponentialCorridorTiling, the exponential width corridor tiling problem. □

6.2.4 Deterministic Regular Expressions

Theorem 3.6 showed that there are designs with a perfect yet not DRE-expressible typing. We show next that it there even designs that have local typings, but none of the local typings is definable by deterministic regular expressions.

THEOREM 6.13. *There are designs $D = (R, w)$, where R is a DRE, such that there exists a local typing for D, but there is no DRE-definable local typing for D.*

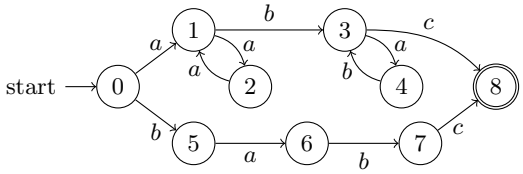
PROOF. Let $w = f_1 abf_2$ and $R = a(aa)^* b(ab)^* c + babc$. Notice that R is a deterministic regular expression. The minimal DFA A for $L(R)$ is shown in Figure 5(a).

We claim that D only has the local typing $(R_1), L(R_2))$ with $R_1 = (aa)^*(ab)^* + b$ and $R_2 = c$. Indeed, since $L(R)$ contains the string $babc$ and $w = f_1 abf_2$, we have, for every local typing $\tau = (L_1, L_2)$, that $c \in L_2$. Furthermore, L_2 cannot contain any other string than c: suppose, towards a contradiction, that L_2 contains $u \neq c$. But then $babu \in w(\tau') \setminus L(R)$ which contradicts that τ' is a local typing. Hence, L_2 must be $\{c\}$.

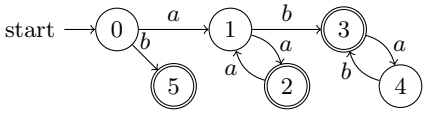
As $L_2 = \{c\}$ and τ is local, we immediately get $L_1 = \{v \mid v \cdot abc \in L(R)\}$ and thus $L_1 = L((aa)^*(ab)^* + b)$. The minimal DFA A_1 for L_1 is shown in Figure 5(b). It can be shown that $L(R_1)$ is not definable by a DRE. □

Since the typing τ in the proof of Theorem 6.13 is in strong A-normal form we immediately get the following corollary.

COROLLARY 6.14. *Not every A-NF typing, where A is the minimal DFA for the language $L(R)$ of a DRE, is DRE-definable.*



(a) Minimal DFA A for $L(R)$.



(b) Minimal DFA A_1 for $L(R_1)$.

Figure 5: Minimal DFAs for the regular expressions R and R_1 in the proof of Theorem 6.13.

Due to Corollary 6.14, the upper bound for ∃−LOC(DFA) cannot be transferred to ∃−LOC(DRE), as it depends on A-NF typings. It could be possible, that there is a DRE-definable local typing τ for a design $D = (R, w)$, where R is a DRE, but the induced A-NF typing is not DRE-definable.

We also show that, unlike in the DFA case, ∃−LOC(DRE) is different from ∃−ML(DRE). Especially Theorem 2.3 does not hold, if the term regular is replaced by DRE defineable regular.

THEOREM 6.15. *There is a design $D = (R, w)$ where R is a DRE such that D has a local DRE-definable typing, but no maximal local DRE-definable typing.*

PROOF. In the proof of Theorem 3.6, we already showed, that the perfect typing $\tau = (L(\Omega_1), L(\Omega_2))$ for the design D_2 from Example 3.2 is not expressible by DREs. As τ is a perfect typing for D_2, there can be no other (possibly DRE-definable) maximal local typing for D_2. However, the DRE-definable typing $\tau_2 = (a(ba)^*, (bc)^* d)$ is a local typing for D_2.

On the other hand, a maximal DRE-definable typing $\tau = (L_1, L_2)$ cannot exist as otherwise $L(\Omega_1) \setminus L_1$ or $L(\Omega_2) \setminus L_2$ would contain a string w and DRE-definable languages are closed under adding a single string (again: Lemma 10 in [4]). □

THEOREM 6.16. Primality *and* StrongPrimality *are* PSPACE*-complete even if the language is given as a DRE.*

LEMMA 6.17. ∃−LOC(DRE) *and* ∃−ML(DRE) *are* PSPACE*-hard, already for designs of the form $(L, f_1 a f_2 a)$.*

Similar as before, the lower bound proof of Lemma 6.17 gives us the inspiration for also proving that ML(DRE) is PSPACE-hard.

LEMMA 6.18. ML(DRE) *is* PSPACE*-hard.*

Similarly as for DFAs, we can prove a more efficient upper bound for ∃−ML(DRE) if the distributed string has only two function calls.

THEOREM 6.19. ∃−ML(DRE) *is in* EXPSPACE *and* ∃−2ML(DRE) *is* PSPACE*-hard and in* EXPTIME.

PROOF. The lower bound is immediate from Lemma 6.17.

For the upper bound we use the following algorithm, which can be implemented to use only exponential space in the general case and exponential time in the case with only 2 function calls.

(1) Compute a DFA A for the given DRE R.

(2) Compute the set T of all local typings in strong A-NF for (A, w).

(3) Compute the set $T_{\max}$ of all maximal typings from T.

(4) Check if there is a DRE-definable typing in $T_{\max}$ □

7. FURTHER WORK

The array for further research directions is wide. From a theoretical point of view, the most interesting question is settling the complexities of $\exists$−LOC(DFA) and $\exists$−LOC(NFA). Solving these questions could also bring new insights in open problems in formal language theory concerning language primality, since factorizing in more than two languages is also of interest there. From a more applied point of view, the most interesting questions undoubtedly concern the deterministic content models (DFAs and DREs). To mention one example: if (maximal) local or perfect typings exist w.r.t. DFAs or DREs, how large will they be? From our normal forms for DFAs, we can only infer exponential upper bounds. Furthermore, since DREs can be exponentially larger than their DFA-representation in general [6, 4], this question becomes even more intriguing for DREs. Furthermore, since not every regular language is DRE-definable, can we be happy with approximations? Finally, it may also be interesting to investigate the typing questions for subclasses of regular expressions that are found commonly in XML schemas on the Web.

8. REFERENCES

[1] S. Abiteboul, G. Gottlob, and M. Manna. Distributed XML design. In *ACM PODS*, pages 247–258, 2009.

[2] S.V. Avgustinovich and A. Frid. A unique decomposition theorem for factorial languages. *Int. J. of Algebra and Comput.*, 15:149–160, 2005.

[3] Sebastian Bala. Regular language matching and other decidable cases of the satisfiability problem for constraints between regular open terms. In *STACS*, pages 596–607, 2004.

[4] G. J. Bex, W. Gelade, W. Martens, and F. Neven. Simplifying XML Schema: effortless handling of nondeterministic regular expressions. In *ACM SIGMOD*, pages 731–744, 2009.

[5] T. Bray, J. Paoli, C. M. Sperberg-McQueen, E. Maler, and F. Yergeau. Extensible Markup Language XML 1.0 (fifth edition). Technical report, World Wide Web Consortium (W3C), November 2008. W3C Recommendation, http://www.w3.org/TR/2008/REC-xml-20081126/.

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

[7] D. Calvanese, G. De Giacomo, M. Lenzerini, and M.Y. Vardi. Rewriting of regular expressions and regular path queries. *J. Comp. Syst. Sc.*, 64(3):443–465, 2002.

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

[9] J.H. Conway. *Regular Algebra and Finite Machines*. Chapman and Hall, 1971.

[10] J. Czyzowicz, W. Fraczak, A. Pelc, and W. Rytter. Linear-time prime decompositions of regular prefix codes. *Int. J. Found. Comp. Sc.*, 14:1019–1031, 2003.

[11] D. Fallside and P. Walmsley. XML Schema Part 0: Primer (second edition). Technical report, World Wide Web Consortium, October 2004. http://www.w3.org/TR/2004/REC-xmlschema-0-20041028/.

[12] S. Gao, C. M. Sperberg-McQueen, H.S. Thompson, N. Mendelsohn, D. Beech, and M. Maloney. W3C XML Schema Definition Language (XSD) 1.1 part 1: Structures. Technical report, World Wide Web Consortium, April 2009. W3C Recommendation, http://www.w3.org/TR/2009/CR-xmlschema11-1-20090430/.

[13] Y.-S. Han, K. Salomaa, and D. Wood. Prime decompositions of regular languages. In *DLT*, pages 145–155, 2006.

[14] T. Jiang and B. Ravikumar. Minimal NFA problems are hard. *Siam J. Comp.*, 22(6):1117–1141, 1993.

[15] M. Kunc. What do we know about language equations? In *DLT*, pages 23–27, 2007.

[16] W. Martens, F. Neven, and T. Schwentick. Complexity of decision problems for XML schemas and chain regular expressions. *Siam J. Comp.*, 39(4):1486–1530, 2009.

[17] Y. Papakonstantinou and V. Vianu. DTD inference for views of XML data. In *ACM PODS*, pages 35–46, 2000.

[18] A. Salomaa, K. Salomaa, and S. Yu. Length codes, products of languages and primality. In *LATA*, pages 476–486, 2008.

[19] A. Salomaa and S. Yu. On the decomposition of finite languages. In *DLT*, pages 22–31, 1999.

[20] K. Salomaa. Language decompositions, primality, and trajectory-based operations. In *CIAA*, pages 17–22, 2008.

[21] P. van Emde Boas. The convenience of tilings. In A. Sorbi, editor, *Complexity, Logic and Recursion Theory*, volume 187 of *Lecture Notes in Pure and Applied Mathematics*, pages 331–363. Marcel Dekker Inc., 1997.

[22] M. Y. Vardi. An automata-theoretic approach to linear temporal logic. In *BANFF*, pages 238–266, 1995.

[23] W. Wieczorek. An algorithm for the decomposition of finite languages. *Logic J. of the IGPL*, 2009. Appeared on-line August 8, 2009.

[24] S. Yu. Regular languages. In G. Rozenberg and A. Salomaa, editors, *Handbook of Formal Languages*, volume 1, chapter 2. Springer, 1997.