Abstract

A distributed XML document is an XML document that spans several machines. We assume that a distribution design of the document tree is given, consisting of an XML kernel-document \( T_{f_1, \ldots, f_n} \) where some leaves are “docking points” for external resources providing XML subtrees \( \{f_1, \ldots, f_n\} \), standing, e.g., for Web services or peers at remote locations). The top-down design problem consists in, given a type (a schema document that may vary from a DTD to a tree automaton) for the distributed document, “propagating” locally this type into a collection of types, that we call typing, while preserving desirable properties. We also consider the bottom-up design which consists in, given a type for each external resource, exhibiting a global type that is enforced by the local types, again with natural desirable properties. In the article, we lay out the fundamentals of a theory of distributed XML design, analyze problems concerning typing issues in this setting, and study their complexity.

Keywords: Semistructured Data, XML Schemas, Distributed Data, Database Design, Distributed XML

1. Introduction

Context and Motivation. With the Web, information tends to be more and more distributed. In particular, the distribution of XML data is essential in many areas such as e-commerce (shared product catalog), collaborating editing (e.g., based on WebDAV [21]), or network directories [24]. (See also the W3C XML Fragment Interchange Working Group [19].) It becomes often cumbersome to verify the validity, e.g., the type, of such a hierarchical structure spanning several machines. In this paper, we consider typing issues raised by the distribution of XML documents. We introduce “nice” properties that the distribution should obey to facilitate type verification based on locality conditions. We propose an automata-based study of the problem. Our theoretical investigation provides a
starting point for the distributed validation of tree documents (verification) and
for selecting a distribution for Web data (design). In general, it provides new
insights in the typing of XML documents.

A distributed XML document $T_{[t_1..t_n]}$ is given by an XML kernel-document
$T_{[f_1,...,f_n]}$, that is stored locally at some site, some of which leaves (the docking
points) refer to external resources, here denoted by $f_1,...,f_n$, that provide the
additional XML data $t_1..t_n$ to be attached, respectively, to $T$. For simplicity,
each node playing the role of docking point is called a function-node and it is
labeled with the resource that it refers.

The extension $\text{ext}_T(t_1..t_n)$ of $T$ is the whole XML document obtainable from
the distributed document $T_{[t_1..t_n]}$ by replacing the node referring resource $f_i$ with
the forest of XML trees (in left-to-right order) directly connected to the root of
$t_i$, for each $i$ in $[1..n]$.

![Figure 1: A distributed XML document for the National Consumer Price Index where both its kernel and some of its remote XML (sub)documents have been hightailed.](image)

Typically, a global designer first chooses a specific language for constraining
the documents of interest. The focus in this paper is on “structural constraints”.
Clearly, one could also consider other constraints such as key and referential
constraints. So, say the designer has to specify documents using DTDs. Then
he specifies a kernel document $T_{[f_1,...,f_n]}$ together with either:

- **bottom-up design**: types $\tau_i$ for each $f_i$;
- **top-down design**: a global type $\tau$.

---

1See [http://epp.eurostat.ec.europa.eu](http://epp.eurostat.ec.europa.eu)
2See [http://ec.europa.eu/eurostat](http://ec.europa.eu/eurostat)
In the bottom-up case, we are interested in the global type that results from each local source enforcing its local type. Can such typing be described by specific type languages?

In the top-down case, we would like the extension of \( T \) to satisfy \( \tau \). The issue is “Is it possible to enforce it using only local control?” In particular, we would like to break down \( \tau \) into local types \( \tau_i \) that could be enforced locally. More precisely, we would like to provide each \( f_i \) with a typing \( \tau_i \) guaranteeing that (i) if each \( f_i \) verifies its type, then the global type is verified (soundness), and (ii) the typing \( \tau_1..\tau_n \) is not more restrictive than the global type (completeness). We call such a typing local typing. We both study (maximal) local typings and an even more desirable notion, namely “perfect typings” (to be defined).

To conclude this introduction, we next detail the Eurostat example. We then present a formal overview of the paper (which may be skipped in a first reading.) Finally, we survey related works.

**Working Example.** Before mentioning some related works and concluding this section, we further illustrate these concepts by detailing our Eurostat example.

The NCPI is a document containing consumer price data for each EC country. We assume that the national data are maintained in local XML repositories by each country’s national statistics bureau (INSEE for France, Statistik for Austria, Istat for Italy, UK Statistics Authority, and so on). Each national data set is under the strict control of its respective statistics bureau. The kernel document \( T_0 \) is maintained by Eurostat in Luxembourg and has a docking point for each resource \( f_i \) located in a particular country. In addition, \( T_0 \) contains average data for the entire EU zone. Figure 2 shows a possible extension of \( T_0 \), where the actual data values are omitted.

![Figure 2: The extension of a possible distributed document having kernel \( T_0 \), and complying with the whole structure showed in Figure 1.](image)

We first assume that Eurostat specifies the global type \( \tau \) for the distributed NCPI document, where \( \tau \) is given by the DTD document shown in Figure 3. (In the following, we adopt a more succinct notation for types where the content model of an element name is either left undefined if it is solely “\#PCDATA”, or defined by a rule of the form “\text{index} \rightarrow \text{value, year}”, otherwise.) Briefly, DTD \( \tau \) requires that each possible extension \( \text{ext}_{T_0}(t_1..t_n) \) consists of a subtree containing average data for Goods (such as food, energy, education, and so on).
Each *Good* item is evaluated in different years by means of an *index*. Moreover, \( \text{ext}_{T_0}(t_1..t_n) \) may contain a forest of *nationalIndex*, namely indexes associated to goods in precise countries.

```
<!ELEMENT eurostat (averages, nationalIndex*)>
<!ELEMENT averages (Good, index+)
+
>
<!ELEMENT nationalIndex (country, Good, (index | value, year))>
<!ELEMENT index (value, year)>
<!ELEMENT country (#PCDATA)>
<!ELEMENT Good (#PCDATA)>
<!ELEMENT value (#PCDATA)>
<!ELEMENT year (#PCDATA)>
```

Figure 3: W3C DTD \( \tau \)

To comply with different national databases, two different formats are allowed: (country, Good, index) or (country, Good, value, year). It is easy to see that the pair \( \langle \tau, T_0 \rangle \) allows a local typing (see Figure 4) that is even perfect (so, can be obtained by the algorithm shown in Section 4), as we will clarify in the next section.

```
root_i \rightarrow \text{nationalIndex}^*

\text{nationalIndex} \rightarrow \text{country, Good, (index | value, year)}
\text{index} \rightarrow \text{value, year}
```

Figure 4: Type \( \tau_i \) (1 \( \leq i \leq n \)) in the perfect typing for the top-down design \( \langle \tau, T_0 \rangle \)

Suppose now that a designer defined instead the DTD \( \tau' \) shown in Figure 5 as global type. The pair \( \langle \tau', T_0 \rangle \) would be a bad design since \( \tau' \) imposes to all countries to adopt the same format for their indexes (natIndA or natIndB). But this represents a constraint that cannot be controlled locally. Indeed, this new design does not admit any local typing. The nice locality properties of designs are obvious in such simplistic examples. However, when dealing with a large number of peers with very different desires and complex documents, the problem rapidly starts defeating human expertise. Consider, for instance, the type \( \tau'' \)

```
eurostat \rightarrow \text{averages, (natIndA* | natIndB*)}
\text{averages} \rightarrow \text{(Good, index+)+}
\text{natIndA} \rightarrow \text{country, Good, index}
\text{natIndB} \rightarrow \text{country, Good, value, year}
\text{index} \rightarrow \text{value, year}
```

Figure 5: Type \( \tau' \)

defined in Figure 6 and the kernel \( T_1 = \text{eurostat}(f_1, \text{nationalIndex}(f_2), f_3) \) containing only three function calls. Even if this design is as small as \( \langle \tau, T_0 \rangle \), it already starts to become hard to manage with no automatic technique. Here, natIndA and natIndB are different specializations of *nationalIndex* elements.
(note that, as detailed in Section 2.2, this feature requires schema languages more expressive than DTDs), while all other elements have no specialization.

\begin{align*}
eurostat & \rightarrow \text{averages}, (\text{natIndA}, \text{natIndB})^+ \\
\text{averages} & \rightarrow (\text{Good}, \text{index}^+)^+ \\
\text{natIndA} & \rightarrow \text{country}, \text{Good}, \text{index} \\
\text{natIndB} & \rightarrow \text{country}, \text{Good}, \text{value}, \text{year} \\
\text{index} & \rightarrow \text{value}, \text{year}
\end{align*}

Figure 6: Type $\tau''$

In this case, it is not as easy as before to state that the new design has no perfect typing and exactly the two maximal local typings shown below (only the content models of the roots are specified). This is mainly because the functions in $T_1$ have different depth, but also due to specializations.

\begin{align*}
\tau''_{1.1}: & \ \text{root}_1 \rightarrow \text{averages}, (\text{natIndA}, \text{natIndB})^* \\
\tau''_{2.1}: & \ \text{root}_2 \rightarrow \text{country}, \text{Good}, \text{index} \\
\tau''_{3.1}: & \ \text{root}_3 \rightarrow \text{country}, \text{Good}, \text{value}, \text{year}, (\text{natIndA}, \text{natIndB})^* \\
\tau''_{1.2}: & \ \text{root}_1 \rightarrow \text{averages}, (\text{natIndA}, \text{natIndB})^*, \text{natIndA} \\
\tau''_{2.2}: & \ \text{root}_2 \rightarrow \text{country}, \text{Good}, \text{value}, \text{year} \\
\tau''_{3.2}: & \ \text{root}_3 \rightarrow (\text{natIndA}, \text{natIndB})^*
\end{align*}

The techniques developed in this paper are meant to support experts in designing such distributed document schemas.

Overview of Results. We next precise the formal setting of the paper and its results. From a formal viewpoint, we use Active XML terminology and notation for describing distributed documents \[1\].

Not surprisingly, our results depend heavily of the nature of the typing that is considered. For types, we consider abstract versions of the conventional typing languages \[6, 29, 33, 35\], namely $R$-DTDs (for W3C DTDs), $R$-SDTDs (for W3C XSD), and $R$-EDTDs (for regular tree grammars such as Relax-NG) where $R$ (varying among $n$FAs, $d$FAs, $n$REs, and $d$REs, namely automata and regular expressions both nondeterministic and deterministic) denotes the formalism for specifying content models.

As a main contribution, we initiate a theory of local typing. We introduce and study three main notions of locality: local typing, maximal local typing, and perfect typing. For a given XML schema language $S$, we study the following verification problems:

- Given an $S$-typing for a top-down $S$-design, determine whether the former is local, maximal local, or perfect. We call these problems LOC$_S$, ML$_S$, and PERF$_S$, respectively;
- Given a top-down $S$-design, establish whether a local, maximal local, or perfect $S$-typing does exist (and, of course, find them). We call these problems $\exists$-LOC$_S$, $\exists$-ML$_S$, and $\exists$-PERF$_S$, respectively;

5
Given a bottom-up $S$-design, establish whether it defines an $S$-type. The problem is called $\text{CONS}_S$.

The analysis carried out in this paper provides tight complexity bounds for some of these problems. In particular, for bottom-up designs, we prove that $\text{CONS}_S$ is:

- decidable in constant time for $\mathcal{R}$-EDTDs, for each $\mathcal{R}$;
- $\text{PSPACE}$-complete both for $\mathcal{R}$-DTDs and $\mathcal{R}$-SDTDs, in general;
- $\text{PSPACE}$-hard with an $\text{EXPTIME}$ upper bound for $\mathcal{R}$-EDTDs and $\mathcal{R}$-SDTDs.

For top-down designs, after showing that the problems for trees can be reduced to problems on words, we specialize the analysis to the case of $\mathcal{R} = \text{nFA}$. In particular:

- $\text{LOC}_S$, $\text{ML}_S$, $\text{PERF}_S$, and $\exists\text{-PERF}_S$ are $\text{PSPACE}$-complete when $S$ stands for $\text{nFA}$-DTD or $\text{nFA}$-SDTD, and $\text{LOC}_S$ is $\text{EXPTIME}$-complete for $\text{nFA}$-EDTDs;
- $\exists\text{-LOC}_S$ and $\exists\text{-ML}_S$ are $\text{PSPACE}$-hard with an $\text{EXPSPACE}$ upper bound when $S$ stands for $\text{nFA}$-DTD or $\text{nFA}$-SDTD;
- the remaining problems are $\text{EXPTIME}$-hard with either $\text{coNEXPTIME}$ or $2\text{-EXPSPACE}$ upper bounds.

Related Work. Distributed data design has been studied quite in depth, in particular for relational databases [13, 34]. Some previous works have considered the design of Web applications [12]. They lead to the design of Web sites. The design there is guided by an underlying process. It leads to a more dynamic notion of typing, where part of the content evolves in time, e.g., creating a cart for a customer. For obvious reasons, distributed XML has raised a lot of attention recently. Most works focused on query optimization, e.g., [4]. The few that consider design typically assume no ordering or only limited one [9]. This last work would usefully complement the techniques presented here. Also, works on relational database and LDAP design focus on unordered collections.

Even the W3C goes in this direction with a working group on XML Fragment Interchange [19]. The goal is to be able to process (e.g., edit) document fragments independently. Quoting the W3C Candidate Recommendation: “It may be desirable to view or edit one or more [fragments] while having no interest, need, or ability to view or edit the entire document.” This is clearly related to the problem we study here. Finally, the concept of distributed documents, as defined in this paper, is already implemented in Active XML, a declarative framework that harnesses web services for data integration, and is put to work in a peer-to-peer architecture [1, 2]. Moreover, XML documents, XML schemas, and formal languages have been extensively studied and, although all the problems treated in this paper are essentially novel, the theoretical analysis has got

---

3Lightweight Directory Access Protocol (LDAP) is a set of open protocols used to access centrally stored information over a network.

4Consider that, as highlighted in [31], an interesting problem in Formal Language Theory open for more than ten years, named Language Primality, is essentially a special case of our problem $\exists\text{-LOC}_S[\text{FA}]$. The complexity of Primality has been also settled in [31].
benefit from a number of existing works. Classical results about formal (string and regular) languages come from \cite{20, 23, 25, 26, 32, 36, 37, 42} and in particular, those about state complexity of these languages can be found in \cite{22, 43}, those about one-unambiguous regular languages in \cite{7, 11}, and those about alternating finite state machines in \cite{17, 42}. Finally, regarding XML documents and schemas, our abstract presentation builds on ten years of research in this field. In particular, it has been strongly influenced by document typings studied in \cite{4, 8, 10, 14, 15, 27, 30, 33, 35, 38} and results on them presented there.

Structure of the Paper. This concludes the introduction. The remaining of the paper is organized as follows. Section 2 fixes some preliminary notation, formally introduces our notions of type, distributed XML document, and defines the decision problems studied. It also provides an overview of the results. Section 3 considers the bottom-up design. Section 4 presents basic results regarding the top-down design. Sections 5 and 6 present the main results for the word case. Section 7 completes the complexity analysis. Section 8 concludes and mentions possible areas for further research.

2. General setting

In this paper, we use a widespread abstraction of XML documents and XML Schemas focusing on document structure \cite{3, 29, 33, 35}, and Active XML terminology and notation for describing distributed documents \cite{1, 2}. In particular, for XML Schemas we will consider families of tree grammars (called \textit{R-DTDs}, \textit{R-SDTDs}, and \textit{R-EDTDs}) each of which allows different formalisms for specifying content models \((R\) vary among \textit{nFAs}, \textit{dFAs}, \textit{nREs}, and \textit{dREs}, respectively, nondeterministic automata, deterministic automata, regular expressions, and deterministic regular expressions). This, could be surprising at first sight because the W3C standards impose stricter limitations. However, as we will informally motivate later, (and has been formally proved in \cite{31}), some of the problems we define and analyze here, have the same complexity independently of whether we use deterministic or nondeterministic string-automata, or even deterministic regular expressions. Informally, it can be observed that the document distribution often erases the benefits of determinism. For this reason, and because this paper intends to be a first fundamental study of XML distribution, we include in our analysis different possibilities for schema languages, even if for some problem we only analyze the most general case \((R\) is set to \textit{nFAs}) in order to delimit its complexity. Moreover, the typing problems we study hint at the possibility that there could be interesting real world applications (all distributed applications that involve the management of distributed data, such as data integration from databases and other data resources exported as Web services, or managing active views on top of data sources) where W3C recommendations are too strict and thus unsuitable in the context of distributed XML documents.
2.1. Preliminaries

In this paper, we use also the following notation. We always denote, by \( \Sigma \), a (finite) alphabet; by \( \varepsilon \), the empty string; by \( \emptyset \), the empty language; by \( \cdot \), the binary relation of concatenation on \( \Sigma^* \) and by \( \circ \), its extension on \( 2^{\Sigma^*} \); by \( \mathcal{A} \), an automaton for defining a string-language or tree-language over \( \Sigma \); by \( \mathcal{R} \), a formalism for defining string languages; by \( \mathcal{S} \), a formalism for defining tree languages; by \( \tau \), an \( R \)-type or an \( S \)-type (a concrete formal structure defining, respectively, a string languages or a tree languages, such as a regular expression or an XML schema document) over \( \Sigma \); by \( [\tau] \), the language defined by \( \tau \).

2.1.1. XML Documents

An XML document can be viewed, from a structural point of view, as a finite ordered, unranked tree (hereafter just a tree) \( t \) with nodes labeled over a given alphabet \( \Sigma \). The topmost node in \( t \) is denoted by root(\( t \)), while for any node \( x \) of \( t \), we denote by

- \( \text{parent}_t(x) \) the (unique) parent node of \( x \) (if node \( x \) is not the root);
- \( \text{children}_t(x) \) is the sequence of children (possibly empty) of \( x \) in left-to-right order;
- \( \text{tree}_t(x) \) the subtree of \( t \) rooted at \( x \);
- \( \text{lab}_t(x) \in \Sigma \) the label of \( x \);
- \( \text{anc-str}_t(x) \in \Sigma^+ \) is the sequence of labels of the path from the root of \( t \) to \( x \);
- \( \text{child-str}_t(x) \in \Sigma^* \) the labels of the children of \( x \) in left-to-right order.

In particular, if \( \text{child-str}_t(x) = \varepsilon \), then \( x \) is called a leaf node. The size of \( t \), denoted by \( ||t|| \), is the number of its nodes. Also in these predicates we may omit the subscript \( t \) when it is clear from the context.

2.1.2. Regular String Languages

A nondeterministic finite state machine (nFA) over \( \Sigma \) is a quintuple \( \mathcal{A} = \langle K, \Sigma, \Delta, q_0, F \rangle \) where \( K \) are the states, \( q_0 \in K \) is the initial state, \( F \subseteq K \) are the final states, and \( \Delta \subseteq K \times (\Sigma \cup \{\varepsilon\}) \times K \) is the transition relation. Each triple \((q, \alpha, q') \in \Delta \) is called a transition of \( \mathcal{A} \). Sometimes the notation \( q' \in \Delta(q, \alpha) \), where \( \Delta \) is seen as a function from \( K \times (\Sigma \cup \{\varepsilon\}) \) to \( 2^K \), is more convenient. By \( \Delta^* \subseteq K \times \Sigma^* \times K \) we denote the extended transition relation defined as the reflexive-transitive closure of \( \Delta \), in such a way that \((q, w, q') \in \Delta^* \) iff there is a sequence of transitions from \( q \) to \( q' \) recognizing string \( w \). The set of strings \( \{w \in \Sigma^* : \Delta^*(q_0, w) \in F\} \) is the language defined by \( \mathcal{A} \). Such machines can be combined in various ways (see [22] for a comprehensive analysis). In particular, \( \mathcal{A} \) denotes the complement of \( \mathcal{A} \), and defines the language \( \Sigma^* - [\mathcal{A}] \).

Given two nFAs \( \mathcal{A}_1 \) and \( \mathcal{A}_2 \), we denote by \( \mathcal{A}_1 \cdot \mathcal{A}_2 \), \( \mathcal{A}_1 \cup \mathcal{A}_2 \), \( \mathcal{A}_1 \cap \mathcal{A}_2 \), and \( \mathcal{A}_1 - \mathcal{A}_2 \), the nFA defining \([\mathcal{A}_1] \circ [\mathcal{A}_2] \), \([\mathcal{A}_1] \cup [\mathcal{A}_2] \), \([\mathcal{A}_1] \cap [\mathcal{A}_2] \), and \([\mathcal{A}_1] - [\mathcal{A}_2] \), respectively (operators \( \cdot \) and \( \circ \) are often omitted). Also, for a set \( \mathcal{A} = \{ \mathcal{A}_1, \ldots, \mathcal{A}_m \} \) of nFAs, we often write \( \cap \mathcal{A} \) (or \( \cup \mathcal{A} \)) instead of \( \mathcal{A}_1 \cap \ldots \cap \mathcal{A}_m \) (or \( \mathcal{A}_1 \cup \ldots \cup \mathcal{A}_m \)).
A deterministic finite automaton (dFA) over $\Sigma$ is an nFA where $\Delta$ is a function from $K \times \Sigma$ to $K$.

A (possibly nondeterministic) regular expression (nRE or also regex, for short) $r$ over $\Sigma$ is generated by the following abstract syntax:

$$r ::= \varepsilon \mid \emptyset \mid a \mid (r \cdot r) \mid (r + r) \mid r? \mid r^+ \mid r^*$$

where $a$ stands generically for the elements of $\Sigma$. When it is clear from the context, we avoid unnecessary brackets or the use of $\cdot$ for concatenation. The language $[r]$ is defined as usual.

A deterministic regular expression (dRE) $r$ is an nRE with the following restriction. Let us consider the regex $\tilde{r}$ built from $r$ by replacing each symbol $a \in \Sigma$ with $\tilde{a}^i$ where $i$ is the position from left-to-right of $a$ in $r$. By definition, $r$ is a dRE if there are no strings $w\tilde{a}^i u$ and $w\tilde{a}^j v$ in $[\tilde{r}]$ such that $i \neq j$. The language $[r]$ of a dRE $r$ is called one-unambiguous [11].

A cartesian product of $n$ finite sets is called a “box” [41]. More precisely, fix a positive number $n$. Let $\Sigma$ be an alphabet. A box $B$ over $\Sigma$ is any language of the form $\Sigma_1 \ldots \Sigma_n$ where $n$ is its width, and $\Sigma_i \subseteq \Sigma$ for each $i$ in $[1..n]$. Clearly, each box is a regular language as it is a finite one.

2.1.3. Regular Tree Languages

A nondeterministic Unranked Tree Automaton (nUTA) is a quadruple $A = \langle K, \Sigma, \Delta, F \rangle$ where $\Sigma$ is the alphabet, $K$ is a finite set of states; $F \subseteq K$ is the set of final states; $\Delta$ is a function mapping pairs from $(K \times \Sigma)$ to nFAs over $K$. A tree $t$ belongs to $[A]$ if and only if there is a mapping $\mu$ from the nodes of $t$ to $K$ such that (i) $\mu(\text{root}(t)) \in F$, and (ii) for each node $x$ of $t$, either $\varepsilon$ or $\mu(\text{children}(x))$ belongs to $[\Delta(\mu(x), \text{lab}(x)))]$ according to whether $x$ is a leaf-node or not, respectively.

A bottom-up-deterministic Unranked Tree Automaton (dUTA) over $\Sigma$ is an nUTA where $\Delta$ is a function from $(K \times \Sigma)$ to dFAs over $K$ in such a way that $[\Delta(q, a)] \cap [\Delta(q', a)] = \emptyset$ for each $q \neq q'$.

2.1.4. Known decision problems

In this section we recall some well known decision problems.

**Definition 1.** \textsc{equiv}$_{[S]}$ is the following decision problem. Given two $S$-types, do they define the same language? \hfill \square

In particular, whenever we consider two $R$-types instead of $S$-types, we still denote by \textsc{equiv}$_{[R]}$ the equivalence problem defined exactly as above.

**Definition 2.** \textsc{one-unamb}$_{[R]}$ is the following decision problems. Given a regular language $L$ specified by an $R$-type, is $L$ one-unambiguous? \hfill \square
2.2. Types

As already mentioned, we consider abstractions of the most common XML Schemas by allowing regular languages, specified by possibly different formalisms for defining content models. More formally, let $\mathcal{R}$ be a mechanism for describing regular languages (\text{nFAs}, \text{dFAs}, \text{nREs}, \text{dREs}, or even others). We want to define and computationally characterize the problems regarding Distributed XML design in a comparative analysis among the three main actual formalisms for specifying XML schema documents: \text{W3C DTDs}, \text{W3C XSD} and Regular Tree Grammars (like Relax-NG). For each of these schema languages, we adopt a class of abstractions that we call $\mathcal{R}$-\text{DTDs}, $\mathcal{R}$-\text{SDTDs}, and $\mathcal{R}$-\text{EDTDs}, respectively, where $\mathcal{R}$ is the particular mechanism for defining content models. We show that a number of properties do not depend on the choice of $\mathcal{R}$ (or even of $\mathcal{S}$) and for some complexity results we focus our analysis to the case of \text{nFAs}. Before that, we summarize in Table 1 the relevance of the different tree grammars.

<table>
<thead>
<tr>
<th>Schema language</th>
<th>Previously introduced formalism</th>
<th>Our abstraction</th>
</tr>
</thead>
<tbody>
<tr>
<td>\text{W3C DTDs}</td>
<td>\text{DTDs and ltds}</td>
<td>\text{dRE-EDTDs}</td>
</tr>
<tr>
<td>\text{W3C XSD}</td>
<td>\text{Single-Type Tree Grammars and single-type EDTDs}</td>
<td>\text{dRE-SDTDs}</td>
</tr>
<tr>
<td>\text{Relax NG}</td>
<td>\text{unranked regular tree languages (specialized ltds and EDTDs)}</td>
<td>\text{nRE-EDTDs}</td>
</tr>
</tbody>
</table>

2.2.1. \text{R-DTD} types

The following definition generalizes definitions considered in the literature such as ltds \cite{6,35} or DTDs \cite{29,30}, and defined for analyzing the properties of \text{W3C Document Type Definitions}. As we marry these views, we define the following class of abstractions capturing all of them.

\textbf{Definition 3.} An $\mathcal{R}$-\text{DTD} is formalized as a triple $\tau = \langle \Sigma, \pi, s \rangle$ where

\begin{itemize}
  \item $\Sigma$ is an alphabet (the element names);
  \item $\pi$ is a function mapping the symbols of $\Sigma$ to $\mathcal{R}$-types still over $\Sigma$;
  \item $s \in \Sigma$ is the start symbol.
\end{itemize}

A tree $t$, having labels over $\Sigma$ belongs to $[\tau]$ if and only if: $\text{lab}($\text{root}(t)) = s$ and $\text{child-str}(x) \in [\pi(\text{lab}(x))]$, for each node $x$ of $t$. For a given element name $a$, the regular language $[\pi(a)]$, associated to $a$, is usually called the \textbf{content model} of $a$. \hfill $\square$
Notice that, due to the above definition, $\mathcal{R}$-DTDs with useless element names, or even defining the empty language, do exist. This is because the above definition allows to specify $\mathcal{R}$-DTDs that are, in a sense, “not reduced” (think about finite automata with unreachable states). Since it is much more convenient to deal with types that are not effected by these drawbacks, after giving some more definition, we formalize the notion of reduced types.

We introduce the dFA dual$(\tau)$. It is the language consisting of the set of paths from the root to a leaf in trees in $[\tau]$ and it is in some sense the vertical language of $\tau$.

**Definition 4.** Let $\tau = (\Sigma, \pi, s)$ be an $\mathcal{R}$-DTD. We build from $\tau$ the dual dFA dual$(\tau) = (K, \Sigma, \delta, q_0, F)$ as follows:

- $K = \{q_0\} \cup \{q_a : a \in \Sigma\}$;
- $\delta(q_0, s) = q_s$;
- for each $a, b \in \Sigma$, $\delta(q_a, b) = q_b$ iff $b$ appears in the alphabet of $\pi(a)$;
- $q_a \in F$ iff $\varepsilon \in [\pi(a)]$.

Before defining a set of conditions ensuring that all the content models of a given $\mathcal{R}$-DTD $\tau$ are well defined and have no redundancy w.r.t. the language $[\tau]$, we mark the states of dual$(\tau)$ (in a bottom-up style) as follows:

1. Mark each final state of dual$(\tau)$ as bound;
2. For each non-bound state $q_b$, consider the set $\Sigma_b \subseteq \Sigma$ where $\delta(q_b, a)$ is bound iff $a \in \Sigma_b$. If $[\pi(b)] \cap \Sigma_b^+ \neq \emptyset$, then mark also $q_b$ as bound;
3. Repeat step 2 until no more states can be marked.

**Definition 5.** Let $\tau$ be an $\mathcal{R}$-DTD. We say that $\tau$ is reduced iff

- Each state of dual$(\tau)$ is in at least a path from $q_0$ to a final state in $F$;
- Each state of dual$(\tau)$ is bound;
- [dual$(\tau)$] is nonempty.

We consider only reduced $\mathcal{R}$-DTDs where, by the previous definition, it is clear that $[\tau] \neq \emptyset$. Note that for a given $\mathcal{R}$-DTD $\tau$, it is very easy to build dual$(\tau)$ and for each “unprofitable” state $q_a$:

- remove the element name $a$ from $\Sigma$;
- remove the rule $\pi(a)$ from $\pi$;
- modify the rules containing $a$ in their content models (using standard regular language manipulation) to produce only words not containing $a$ (see [28], for more details.)

Finally, we notice that only the last step of the reducing algorithm may depend on the choice of $\mathcal{R}$. Clearly, an $\mathcal{R}$-DTD and its reduced version describe the same language.

From a theoretical point of view, $\mathcal{R}$-DTDs do not express more than the local tree languages [32]. In particular, nFA-DTDs, dFA-DTDs and nRE-DTDs exactly
capture this class of languages while dRE-DTDs are less expressive \[33, 35\]. Nevertheless, the last class of types (using deterministic regular expressions \[11\] and that does not capture all the local tree languages) is, from a structural point of view, the closest to W3C DTDs.

In this paper, for a given R-DTD where \(R\) stands for dFAs or nFAs (for shortness, w.l.o.g., and only in examples) we often specify \(\pi\) as a function that maps \(\Sigma\)-symbols to \(\Sigma\)-nREs (recall that any regular expression of size \(n\) can be transformed into an equivalent \(\varepsilon\)-free nFA with \(O(n \log^2 n)\) transitions in time \(O(n \log^2 n)\) \[20, 23\].)

Finally, an example of dRE-DTD is \(\tau_1 = \langle \{s_1, c\}, \pi_1, s_1 \rangle\) with \(\pi_1(s_1) = c^*\) and \(\pi_1(c) = \varepsilon\). In the rest of the paper, we often omit to specify rules such as \(\pi_1(c) = \varepsilon\); i.e., if no rule is given for a label, nodes with this label are assumed to be (solely) leaves.

2.2.2. \(R\)-SDTD types

The following definition generalizes definitions considered in the literature such as Single-Type Tree Grammars \[33\] or single-type EDTDs \[29\], and defined for analyzing the properties of W3C XML Schema Definitions. Also here, we define a class of abstractions capturing all of them.

**Definition 6.** An \(R\)-SDTD (standing for single-type extended R-DTD) is a quintuple \(\tau = (\Sigma, \tilde{\Sigma}, \pi, \tilde{s}, \mu)\) where

- \(\tilde{\Sigma}\) are the **specialized element names**;
- \(\langle \tilde{\Sigma}, \pi, \tilde{s} \rangle\) is an \(R\)-DTD on \(\tilde{\Sigma}\) and denoted by \(\text{dtd}(\tau)\);
- \(\mu : \tilde{\Sigma} \to \Sigma\) is a mapping from all the specialized element names onto the set of element names. For each \(a \in \Sigma\), we denote by \(\tilde{a}^1, \ldots, \tilde{a}^n\) the distinct elements in \(\tilde{\Sigma}\) that are mapped to \(a\). This set is denoted \(\tilde{\Sigma}(a)\);
- Let \(\text{dual}(\text{dtd}(\tau)) = (K, \tilde{\Sigma}, \delta, q_0, F)\). Build from this dFA the possibly nFA \(\text{dual}(\tau) = (K, \Sigma, \delta, q_0, F)\) where for each \(q, q' \in K\) and \(a \in \Sigma\), \(\delta(q, a) = q'\) iff there is an element \(\tilde{a} \in \tilde{\Sigma}\) such that \(\delta(q, \tilde{a}) = q'\). We require that \(\text{dual}(\tau)\) is a dFA (this captures the single-type requirement). Also in this case, \(\text{dual}(\tau)\) defines the vertical language of \(\tau\).

A tree \(t\), labeled over \(\Sigma\), is in \(\tau\) if and only if there exists a tree \(t' \in [\text{dtd}(\tau)]\) such that \(t = \mu(t')\) (where \(\mu\) is extended to trees). Informally, we call \(t'\) a witness for \(t\). Finally, an \(R\)-SDTD \(\tau\) is **reduced** if and only if \(\text{dtd}(\tau)\) is.

As for R-DTDs, we consider only reduced \(R\)-SDTDs.

From a theoretical point of view, \(R\)-SDTDs are more expressive than \(R\)-DTDs but do not capture the unranked regular tree languages yet.

2.2.3. \(R\)-EDTDs types

The following definition generalizes definitions considered in the literature such as specialized ltds \[1, 33\] or EDTDs \[29\]. Such formalisms (like Relax-NG), from a structural perspective, express exactly the homogeneous unranked regular tree languages and are as expressive as unranked tree automata or Regular Tree Grammars \[10\].
Definition 7. An \( R-\text{EDTD} \) (extended \( R-\text{DTD} \)) \( \tau \) is an \( R-\text{SDTD} \) without the single-type requirement. More formally, the automaton dual(\( \tau \)), built as for \( R-\text{SDTD} \), may be here an \( n\text{FA} \). The language \([\tau]\) is defined as for \( R-\text{SDTDs} \). \( \square \)

2.3. Distributed Documents

In the context of distributed architectures (e.g., P2P architectures), distributed documents (or distributed trees), such as AXML documents, are XML documents that may contain embedded function calls. In particular, a distributed XML document \( T_{t_1..t_n} \) can be viewed as a collection of (classical) XML documents \( t_1..t_n \) brought together by a unique (special) XML document \( T_{f_1...f_n} \), the kernel, some of whose leaf-nodes, called function-nodes, play the role of “docking points” for the external resources \( f_1,...,f_n \). The “activation” of a node of \( T \) having a function as label, say \( f_i \), consists in a call to resource (or function) \( f_i \), the result of which is still an XML document, say \( t_i \). When \( f_i \) is invoked, its result is used to extend the kernel \( T_{f_1...f_n} \). Thus, each docking point connects the peer that holds the kernel and invokes the resource \( f_i \), and the peer that provides the corresponding XML document \( t_i \). For simplicity of notation, for labeling a function-node we use exactly the name of the resource it refers. For instance, the tree \( T_0 = s(a f_1 b(f_2)) \) is a kernel having \( s \) as root, and containing two function-nodes referring the external resource \( f_1 \) and \( f_2 \).

The extension \( \text{ext}_T(t_1..t_n) \) of \( T \) is the whole XML document (without any function at all) obtained from the distributed document \( T_{t_1..t_n} \) by replacing each node referring resource \( f_i \) with the forest of XML trees (in left-to-right order) directly connected to the root of \( t_i \). This process is called materialization. For instance, the extension of kernel \( T_0 \) would be \( s(a c(dd) b(d(e f))) \) in case of resources \( f_1 \) and \( f_2 \) provided trees \( s_1(c(dd)) \) and \( s_2(d(e f)) \), respectively.

An interesting task is to associate a type \( \tau \) (e.g., a \( W3C \) XSD document) to each resource \( f_i \) in such a way that the XML document \( t_i \) returned as answer is valid w.r.t. this type and any materialization process always produces a document \( \text{ext}_T(t_1..t_n) \) valid w.r.t. a given global type \( \tau \) (still specified by the \( W3C \) XSD syntax). A global type and a kernel document represent the (top-down) design of a given distributed architecture. A collection of types associated to the function calls in such a design is called a typing. Given a distributed design, we would like to know whether either a precise typing has some properties or a typing with some properties does exist. But also, we could directly start from a kernel \( T \) and a typing (bottom-up design) and analyze the properties of the tree language consisting in each possible extension \( \text{ext}_T(t_1..t_n) \).

More formally, let \( \Sigma \) and \( \Sigma^f \) be two alphabets, respectively, of element names (such as \( s, a, b, c \), etc.) and function symbols (such as \( f, g \), etc.). A kernel document or kernel tree \( T_{f_1...f_n} \) (or also \( T(f_n) \), with \( f_n \) denoting a sequence\(^5\) of length \( n \)) is a tree over \( (\Sigma \cup \Sigma^f) \) where:

(i) the root is an element node (say \( s_0 \));

\(^5\)We denote a finite sequence of objects \((x_1,...,x_n)\) over an index set \( I = \{1,...,n\} \) by \((x_n)\) and we often omit the specification of the index set \( I \).
(ii) the function nodes $f_1, \ldots, f_n$ are leaf nodes;
(iii) no function symbol occurs more than once.

In particular, for each non-leaf node of $T$, say $x$, the kernel string $\text{child-str}(x)$, with $k \geq 0$ functions, is of the form $w_h f_{h+1} w_{h+1} \ldots f_{h+k} w_{h+k}$ (for some $h$ in $[1..n]$) where $w_i \in \Sigma^*$ for each $i \in \{h, \ldots, h+k\}$, $f_i \in \Sigma_f$ for each $i \in \{h+1, \ldots, k\}$, and $f_i \neq f_j$ for each $i \neq j$.

We next consider its semantics. It is defined by providing a tree for each function-node. In particular, an extension $\text{ext}$ maps each $i$ in $[1..n]$ to a tree $t_i = \text{ext}(f_i)$. The extension $\text{ext}_T(t_1, \ldots, t_n)$ of a kernel $T[t_1, \ldots, t_n]$ is obtained by replacing each $f_i$ with the forest of trees (in left-to-right order) directly connected to the root of $t_i$.

A type $\tau$ for a kernel tree $T$ is one of an $\mathcal{R}$-DTD, $\mathcal{R}$-SDTD, or $\mathcal{R}$-EDTD. Given an extension $t_1, \ldots, t_n$, we say that tree $T[t_1, \ldots, t_n]$ satisfies type $\tau$ if and only if $\text{ext}_T(t_1, \ldots, t_n)$ does. This motivates requirement (iii) to avoid irregularities: For instance, in the kernel $T_1 = s(f f)$ the children of $s$ in any extension of $T_1$ are of the form $w w$ for some word $w$. But since this is not a regular language, the type of $T_1$ cannot be defined by none of the three adopted formalisms. Although we disallow the same function to appear twice, several functions may share the same type. Also, even if for labeling a function-node we use exactly the name of the resource it refers (for simplicity of notation), this does not prohibit a resource to provide two XML subtrees to be attached to the kernel. In fact, different names (function symbols) can be associated to the same resource still preserving extensions from irregularities.

We introduce typings to constrain the types of the function calls of a kernel document. A typing for a kernel tree $T(f_n)$ is a positional mapping from the functions in $(f_n)$ to a sequence $(\tau_n)$ of types (schema documents). Now, as we replace each $f_i$ (in the extensions of $T$) with a forest of XML documents then, for each type $\tau_i$ associated to $f_i$, we actually use a schema document containing an “extra” element name, say $s_i$, being only the label of the root in all the trees in $[\tau_i]$.

**Definition 8.** We denote by $\text{ext}_T(\tau_n)$ the tree language consisting of all possible extensions $\text{ext}_T(t_1, \ldots, t_n)$ where $t_i \models \tau_i$ ($t_i$ is valid w.r.t. $\tau_i$) for each $i$. \hfill \square

**Definition 9.** We denote by $T(\tau_n)$ the nFA-EDTD (or nRE-EDTD) constructed from $T$ and $(\tau_n)$ in the obvious way such that $[T(\tau_n)] = \text{ext}_T(\tau_n)$. \hfill \square

In Section 3.1, we will show precisely how to build $T(\tau_n)$ in polynomial time, prove that the construction is semantically correct, and establish that the size of $T(\tau_n)$ is purely linear in the size of $T$ and $(\tau_n)$. Let us illustrate for now the issues with an example. Observe, for instance, that for the tree $T = s_0(a(b)f_1 a(c))$, no matter which type $\tau_1$ is, there is no $\mathcal{R}$-DTD-typing expressing the language $\text{ext}_T(\tau_1)$. Indeed, this is even the case for $T = s_0(a(b)a(c))$ with no function at all. If we consider the tree $T = s_0(a(f_1)a(f_2))$, then the typing $[\tau_1] = \{s_1(b)\}$, $[\tau_2] = \{s_2(c)\}$ prohibits that $\text{ext}_T(\tau_1, \tau_2)$ is expressible by an $\mathcal{R}$-DTD-type because $[T(\tau_n)] = \{s_0(a(b)a(c))\}$ entailing that the content model of $b$ is non-regular;
while the typing \( \tau_1 = \{ s_1(b) \} \), \( \tau_2 = \{ s_2(b) \} \) allows that, because \( [T(\tau_n)] = \{ s_0(a(b)a(b)) \} \) entailing that all the content models of \( s_0 \), \( a \) and \( b \) are regular languages, \{aa\}, \{b\} and \( \emptyset \), respectively. Such situations motivated Definition 10.

Before concluding this section, we adapt the previous definitions to strings in the straightforward way. (We will often use reductions to strings problems in the paper.) Let \( w(f_n) = w_0f_1w_1\ldots f_nw_n \) be a kernel string. For typing strings, we use \( R \)-types where \( R \in \{ nFA, dFA, nRE, dRE \} \). A typing for \( w(f_n) \) is still a positional mapping from the functions in \( (f_n) \) to a sequence \( (\tau_n) \) of \( R \)-types. By \( \text{ext}_w(\tau_n) \) we still denote the string language consisting of all possible extensions of \( w \), and by \( w(\tau_n) \) the \( nFA \) (or \( nRE \)) constructed from \( w \) and \( (\tau_n) \) is such a way that \( [w(\tau_n)] = \text{ext}_w(\tau_n) \).

We will use in our proofs a generalization to “Boxes”. A kernel box \( B(f_n) = B_0f_1B_1\ldots f_nB_n \) is, here, a finite regular language over \( (\Sigma \cup \Sigma_\ell) \) where \( f_1, \ldots, f_n \) are as above, and each \( B_i \) is a box (of a fixed width) over \( \Sigma \). With \( B(\tau_n) \) we denote the \( nFA \) (or \( nRE \)) constructed from \( B \) and \( (\tau_n) \) is such a way that \( (B(\tau_n)) = \text{ext}_B(\tau_n) \).

2.4. The Typing Problems

In this section, we introduce the notion of distributed XML design, define the design problems that are central to the present work, and give the overview of the complexity results. We consider two different approaches, bottom-up and top-down, according to whether the distributed design, other than a kernel tree, consists of a typing or a target type, respectively.

Definition 10. Let \( S \) be a schema language, and \( T_{f_1,\ldots,f_n} \) be a kernel document. We call \( S \)-design (or just design) one of the following:

- \( D = (\tau_n), T_{f_1,\ldots,f_n} \) where \( (\tau_n) \) is an \( S \)-typing. This is **bottom-up design**.
- \( D = (\tau, T_{f_1,\ldots,f_n}) \) where \( \tau \) is a (target) \( S \)-type. This is **top-down design**.

Intuitively, given a bottom-up design, one would like to find a global type that captures the typing of the global document. On the other hand, given a top-down design, one would like to find types for the local documents that will guarantee the global type.

With the following definition, we start the bottom-up analysis. Notice that the concepts used for bottom-up design will be also useful when we consider top-down design.

Definition 11. Given an \( S \)-design \( D = (\tau_n), T \), the \( S \)-typing \( (\tau_n) \) is **\( S \)-consistent** with \( T \) (simply consistent when \( S \) is understood) if there exists an \( S \)-type \( \tau \) such that \( [\tau] = \text{ext}_T(\tau_n) \), in other words, if \( \text{ext}_T(\tau_n) \) is definable by some \( S \)-type. This problem (deciding whether an \( S \)-typing is \( S \)-consistent with a kernel tree) is called \text{CONS}_{S_i}.
We will denote by \( \text{type}_{T,S}(\tau_n) \), or \( \text{type}_T(\tau_n) \) when \( S \) is understood, the \( S \)-type when it exists such that \( |\text{type}_T(\tau_n)| = \text{ext}_T(\tau_n) \). Notice that if both \( S \) and \( T \) are fixed, then \( \text{type}_T(\tau_n) \) plays the role of a function from the set of all possible \( S \)-typings of length \( n \) to a set of certain \( S \)-types. According to every possible decision-answer of \( \text{cons}_S \) (where \( T \) is now fixed), such a function might be always definable, never, or only for some \( S \)-typing. Finally, the complexity of deciding \( \text{cons}_S \) or computing \( \text{type}_T(\tau_n) \) (with an estimation, w.r.t. \( T(\tau_n) \), of its possible size), may vary considerably due to \( S \).

Table 2 summarizes the complexity results of \( \text{cons}_S \). We vary \( S \) among \( \mathcal{R} \)-DTDs, \( \mathcal{R} \)-SDTDs and \( \mathcal{R} \)-EDTDs, for various kinds of \( \mathcal{R} \). In all cases but \( d\mathcal{R} \), we get tight bounds. For DTDs and SDTDs with \( d\mathcal{R} \), we provide nonmatching lower and upper bounds. The table also shows the size that \( \text{type}_T(\tau_n) \) may have in the worst case. Again this is given precisely for all cases but \( d\mathcal{R} \). For DTDs and SDTDs with \( d\mathcal{R} \), we provide nonmatching bounds.

In the next sections, we systematically analyze the complexity of this problem by varying \( S \) among \( \mathcal{R} \)-DTDs, \( \mathcal{R} \)-SDTDs and \( \mathcal{R} \)-EDTDs, and we will consider \( \text{type}_T(\tau_n) \) for each of these schema languages. We next give an example to illustrate some of the main concepts introduced.

<table>
<thead>
<tr>
<th>( S )-DTDs</th>
<th>( S )-SDTDs</th>
<th>( S )-EDTDs</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \mathsf{nFA} )</td>
<td>( \mathsf{PSPACE}\text{-complete} )</td>
<td>( \mathsf{DTIME}(O(1)) )</td>
</tr>
<tr>
<td>( \Theta(m) )</td>
<td>( \Theta(m) )</td>
<td>( \Theta(m) )</td>
</tr>
<tr>
<td>( \mathsf{nRE} )</td>
<td>( \mathsf{PSPACE}\text{-complete} )</td>
<td>( \mathsf{DTIME}(O(1)) )</td>
</tr>
<tr>
<td>( \Theta(m) )</td>
<td>( \Theta(m) )</td>
<td>( \Theta(m) )</td>
</tr>
<tr>
<td>( \mathsf{dFA} )</td>
<td>( \mathsf{PSPACE}\text{-complete} )</td>
<td>( \mathsf{DTIME}(O(1)) )</td>
</tr>
<tr>
<td>( \Theta(2^m) )</td>
<td>( \Theta(2^m) )</td>
<td>( \Theta(m^2) )</td>
</tr>
<tr>
<td>( \mathsf{dRE} )</td>
<td>( \mathsf{PSPACE}\text{-hard} \rightleftharpoons \mathsf{EXPTIME} )</td>
<td>( \mathsf{PSPACE}\text{-hard} \rightleftharpoons \mathsf{EXPTIME} )</td>
</tr>
<tr>
<td>( \Omega(2^m) \rightleftharpoons \mathcal{O}(2^{2m}) )</td>
<td>( \Omega(2^m) \rightleftharpoons \mathcal{O}(2^{2m}) )</td>
<td>( \Theta(m) )</td>
</tr>
</tbody>
</table>

**Example 1.** Consider the kernel \( T = s_0(a\ f_1\ c\ f_2) \) and the pair \( \tau_1 = (\{s_1, b\}, \pi_1, s_1) \) and \( \tau_2 = (\{s_2, d\}, \pi_2, s_2) \) of \( d\mathcal{R}\)-DTD-types, with \( \pi_1(s_1) = b^* \) and \( \pi_2(s_2) = d^* \). The activation of both \( f_1 \) and \( f_2 \) may return trees \( s_1(bb) \) and \( s_2(d) \), respectively. These trees can be plugged into \( T \) producing the extension \( s_0(ab^*cd^m) \). Now, we have:

\[
\text{type}_T(\tau_1, \tau_2) = (\{s_0, a, b, c, d\}, \pi, s_0)
\]
where \( \pi(s_0) = a^* b^* c^* d^* \) and all the other element names other that \( s_0 \) are leaves. Finally, \((\tau_1, \tau_2)\) is \(\text{dRE-DTD}\)-consistent with \( T \).

We now define the top-down design problems. But before, we introduce some straightforward notation. Let \( \tau \) and \( \tau' \) be two types. We say that:

* \( \tau \equiv \tau' \) (equivalent) if \( [\tau] = [\tau'] \)
* \( \tau \preceq \tau' \) (smaller or equivalent) if \( [\tau] \subseteq [\tau'] \)
* \( \tau < \tau' \) (smaller) if \( [\tau] \subset \tau' \)

and also that, given two typings \((\tau_n)\) and \((\tau_n')\):

* \( (\tau_n) \equiv (\tau_n') \) if \( \tau_i \equiv \tau_i' \) for each \( i \)
* \( (\tau_n) \preceq (\tau_n') \) if \( \tau_i \preceq \tau_i' \) for each \( i \)
* \( (\tau_n) < (\tau_n') \) if \( (\tau_n) \preceq (\tau_n') \) and \( \tau_i < \tau_i' \) for some \( i \)

**Definition 12.** Given an \( S \)-design \( D = (\tau, T) \), we say that a typing \((\tau_n)\) is:

* **sound** if \( \text{ext}_T(\tau_n) \subseteq [\tau] \);
* **maximal** if it is sound, and there is no other sound typing \((\tau'_n)\) s.t. \( (\tau_n) < (\tau'_n) \);
* **complete** if \( \text{ext}_T(\tau_n) \supseteq [\tau] \);
* **local** if \( \text{ext}_T(\tau_n) = [\tau] \), namely if it is both sound and complete;
* **perfect** if it is local, and \( (\tau_n') \preceq (\tau_n) \) for each other sound typing \((\tau'_n)\);
* **\( D \)-consistent** if it is an \( S \)-typing which is \( S \)-consistent as well.

**Remark 1.** It should be clear that for a given \( S \)-design \( D = (\tau, T) \) we could have sound typings that are not \( D \)-consistent. But, note that, it is even possible to have a sound typing where \( T(\tau_n) \) does not define a regular tree language. Consider the design \( D \) where \( T = s_0(f_1) \) and \( \tau = s_0(a^+ b^+) \). Clearly, the typing \([\tau] = \{ s_1(a^n b^n) : n > 0 \} \) is sound but \([T(\tau_1)] \) is not regular. Anyway, we prove in Section \[\[4.1\]\] (for strings, but the results generalizes to trees due to our reductions) that if an \( S \)-design admits a sound typing \( (\tau_n) \), then it also admits a sound \( n\text{FA}-\text{DTD} \)-typing \( (\tau'_n) \) such that \( (\tau_n) \leq (\tau'_n) \).

Also, by definition of maximality, note that for instance, for a given \( \text{dRE-DTD} \)-design \( D \), a \( \text{dRE-DTD} \)-typing \( (\tau_n) \) is not maximal even if there is a sound \( n\text{FA}-\text{DTD} \)-typing \( (\tau'_n) \) for \( D \) such that \( (\tau_n) < (\tau'_n) \). One could have some objection to such a definition. Anyway, Martens et al. \[\[31\]\] proved that whenever the illustrated situation happens, then there is also a \( \text{dRE-DTD} \)-typing \( (\tau''_n) \) such that \( (\tau_n) < (\tau''_n) \).

Clearly, local typings present the advantage of allowing a local verification of document consistency (soundness and completeness by definition). Also, no consistent document is ruled out (completeness). Maximal locality guarantees that in some sense, no unnecessary constraints are imposed to the participants. Finally, perfect typings are somehow the ultimate one can expect in terms of not imposing constraints to the participants. Many designs will not accept a perfect typing. However, there are maximal sound typings which are not local.
This is not surprising as there are designs that have at least a sound typing but do not allow any local at all, and clearly, if there is a sound typing, then there must also exist a maximal sound one. We will see examples that separate these different classes further. But before, we make an observation on \(D\)-consistency and formally state the problems studied in the paper.

Let \(S\) be any schema language among \(R\)-DTDs, \(R\)-SDTDs, and \(R\)-EDTDs, where \(R \in \{nFA, dFA, nRE, dRE\}\). Whenever we consider a top-down \(S\)-design \(D = (\tau, T)\), we require that a typing \(((\tau_n)\) for \(D\) has to be \(D\)-consistent, namely \(T(\tau_n)\) is \(S\)-consistent (it has an equivalent \(S\)-type) and each \(\tau_i\) is an \(S\)-type. In order to verify such a condition, we can exploit the techniques that we have developed for bottom-up design. In particular, it is not hard to see that if \(((\tau_n)\) is not \(S\)-consistent, then it can not be local. Thus, our approach aims at isolating problems concerning locality from those concerning consistency.

**Definition 13.** \(\text{LOC}_{[S]}\), \(\text{ML}_{[S]}\), \(\text{PERF}_{[S]}\) are the following decision problems. Given an \(S\)-design \(D = (\tau, T)\) and a \(D\)-consistent typing \(((\tau_n)\), is \(((\tau_n)\) a local, or maximal local, or perfect typing for \(D\), respectively?

**Definition 14.** \(\exists\text{-LOC}_{[S]}\), \(\exists\text{-ML}_{[S]}\), \(\exists\text{-PERF}_{[S]}\) are the following decision problems. Given an \(S\)-design \((\tau, T)\), does there exist a local, or maximal local, or perfect \(D\)-consistent typing for this design, respectively?

We similarly define the corresponding word problems (\(S\) is simply \(R\)). We have \(\text{LOC}_{[R]}\), \(\text{ML}_{[R]}\), \(\text{PERF}_{[R]}\), \(\exists\text{-LOC}_{[R]}\), \(\exists\text{-ML}_{[R]}\) and \(\exists\text{-PERF}_{[R]}\). Finally, we will use in proofs box versions of the problems, \(\exists\text{-LOC}_{B_{[R]}}, \exists\text{-ML}_{B_{[R]}}, \exists\text{-PERF}_{B_{[R]}}\).

**Remark 2.** In this paper, although we analyze all the three defined schema languages (\(R\)-DTDs, \(R\)-SDTDs, and \(R\)-EDTDs) for top-down designs, after providing reductions from trees to strings, we specialize the analysis to the case of \(R = nFA\). More tractable problems may be obtained by considering deterministic content models or restricted classes of regular expressions [18, 27] as made by Martens et al. [31]. Also, notice that we pay more attention to maximum locality rather than to maximality proper. In fact, for the latter notion, the existence problem is trivial. Moreover, the complexity of the verification problem essentially coincides for both notions. Nevertheless, one could be interested in a maximal sound typing when, for some reason, the design can not be improved and does not admit any local typing. There could be even cases where a local typing does not exist but, there is a unique maximal sound typing comprising any other possible sound typing, a sort of quasi-perfect typing. For instance, the design \(T = s(a f_1)\) and \(\tau = s(ab^* + d)\) has such a property. Our techniques can be easily adapted to these cases, too.

Table 8 gives an overview of the complexity results for the typings problems previously defined. We will see in Section 8 that, for \(R\)-DTDs and \(R\)-SDTDs, each problem on trees is logspace-reducible to a set of problems on strings (thus, it suffices to prove the results in Table 8 for words) and that, for \(R\)-EDTDs, the problems on trees depend on the problems on boxes in a more complex manner.
In particular, row D includes two problems that are actually the same (they only differ if $R = dREs$, as shown in Martens et al. [31]). Each number in brackets refers either to the corresponding statement/proof in the paper (if rounded) or the paper where the particular result has already been proved (if squared).

Table 3: Complexity results in case of top-down design

<table>
<thead>
<tr>
<th></th>
<th>nFAs / nFA-DTDs / nFA-SDTDs</th>
<th>nFA-EDTDs</th>
</tr>
</thead>
<tbody>
<tr>
<td>[A]</td>
<td>LOC</td>
<td>PS\text{SPACE}-complete 5.3</td>
</tr>
<tr>
<td>[B]</td>
<td>ML</td>
<td>in PS\text{SPACE} [31] $\lor$ PS\text{SPACE}-hard 5.4</td>
</tr>
<tr>
<td>[C]</td>
<td>PERF</td>
<td>PS\text{SPACE}-complete 5.9</td>
</tr>
<tr>
<td>[D]</td>
<td>3-LOC/3-ML</td>
<td>PS\text{SPACE}-hard 5.2 in EXP\text{SPACE} [31] $\lor$ 5.11</td>
</tr>
<tr>
<td>[E]</td>
<td>3-PERF</td>
<td>PS\text{SPACE}-complete 5.9</td>
</tr>
</tbody>
</table>

We now present examples that separate the different design properties of typings.

**Example 2.** Let $\tau = \langle \{s, a, b, c\}, \pi, s \rangle$ be an nRE-DTD where $\pi(s) = a^*bc^*$, and $T = s(f_1f_2)$ be a kernel tree. It is easy to see that both $s_1(a^*bc^*)$, $s_2(c^*)$ and $s_1(a^*)$, $s_2(a^*bc^*)$ are local typings as $a^*bc^*c^* \equiv a^*a^*bc^* \equiv a^*bc^*$. In fact, they are also maximal local typings, and so there is no perfect typing for this design. Observe that, for instance, $s_1(a^*)$, $s_2(a^*bc^*)$ is still a local typing that, however, is not maximal because it imposes unnecessary constraints to the local sites. If desired, one could leave them more freedom, e.g., type the first function with $a^*$.

**Example 3.** Let $\tau = s(a^*bc^*)$ be a type and $T = s(f_1f_2)$ be a kernel tree. The typing $s_1(a^*)$, $s_2(c^*)$ is perfect. This has to be an excellent typing since there is no alternative maximal local typing.

**Example 4.** Let $\tau = (ab)^*$ be a type and $T = s(f_1f_2)$ be a kernel tree. The typing $s_1((ab)^*)$, $s_2((ab)^*)$ is a unique maximal local but it is not perfect. Consider, in fact, typing $s_1(a)$, $s_2(b)$. It is sound but $(a, b) \leq ((ab)^*, (ab)^*)$ does not hold. Clearly, a perfect typing cannot exist.

**Example 5.** Let $\tau = (ab)^+$ be a type and $T = s(f_1f_2)$ be a kernel tree. There are three maximal local typings:

$s_1((ab)^*)$, $s_2((ab)^+)$

$s_1((ab)^+a)$, $s_2(b(ab)^*)$

$s_1((ab)^+)$, $s_2((ab)^*)$

according to whether either $s_1$, $s_1(a)$, or none of them may belong to each possible $\text{ext}(f_1)$, respectively.
The following theorem completes the comparison of the properties of typing we study. (Note that its converse is not true by Example 4.)

**Theorem 2.1.** Every perfect typing is unique maximal local.

**Proof.** Consider a perfect typing \((\tau_n)\) for \(T(f_n)\) and \(\tau\). We observe that \((\tau_n)\) is local, by definition. Moreover, by definition, for each other sound (so also local) typing \((\tau'_n)\), we have \((\tau'_n) \leq (\tau_n)\). The typing, \((\tau_n)\) is maximal because there exists no other sound typing \((\tau''_n)\) such that \((\tau_n) < (\tau''_n)\), and it is unique because there is no another local typing \((\tau''_n)\) such that for some index \(i\) and some string \(w\), then \(w \in [\tau''_i]\) but \(w \notin [\tau_i]\).

3. **Bottom-up design**

In this section, we consider bottom-up design.

3.1. **R-EDTDs typing**

Let \(T(f_n)\) be a kernel and \((\tau_n)\) be an R-EDTD-typing where each \(\tau_i = (\Sigma_i, \hat{\Sigma}_i, \pi_i, \tilde{s}_i, \mu_i)\). We next present the construction of \(T(\tau_n)\), that (to be as general as possible) is an nFA-EDTD. We use the following notations:

1. \(\Sigma_0\) contains the element names in \(T\) (the labels but not the functions);
2. \(\hat{\Sigma}_0\) contains a specialized element name \(\hat{a}_x\), for each \(a \in \Sigma_0\) and each node \(x\) of \(T\) with label \(a\).
3. \(s_0\) is the root of \(T\);
4. \(s_i\) is the root of trees in \([\tau_i]\) for each \(i\).

We also make without loss of generality the following assumptions:

1. \(\hat{\Sigma}_i \cap \hat{\Sigma}_j = \emptyset\), for each \(i, j, i \neq j\). (Note that \(\Sigma_i \cap \Sigma_{j \neq i}\) may be nonempty.)

Consider the nFA-EDTD \(T(\tau_n) = (\Sigma, \hat{\Sigma}, \pi, \tilde{s}_0, \mu)\) defined as follows:

1. \(\Sigma = \Sigma_0 \cup (\Sigma_1 - \{s_1\}) \cup \ldots \cup (\Sigma_n - \{s_n\})\);
2. \(\hat{\Sigma} = \hat{\Sigma}_0 \cup (\hat{\Sigma}_1 - \{\hat{s}_1\}) \cup \ldots \cup (\hat{\Sigma}_n - \{\hat{s}_n\})\);
3. \(\tilde{s}_0 = \hat{s}_0\), where \(x\) is the root of \(T\);
4. \(\mu(\hat{a}) = a\) for each \(\hat{a} \in \hat{\Sigma}\);
5. \(\pi(\hat{a}_x) = \text{nFA}(\{\varepsilon\})\) for each leaf-node \(x\) of \(T\) with label \(a \in \Sigma_0\);
6. \(\pi(\hat{a}_i) = \text{nFA}(\pi_i(\hat{a}_i))\) for each \(\hat{a}_i \in \hat{\Sigma}\) with \(i\) in \([1..n]\);
7. for each node \(x\) of \(T\) with label \(a\) and children \(y_1 \ldots y_p\), we define \(\pi(\hat{a}_x) = \text{nFA}(L_1 \ldots L_p)\) where each language \(L_k\) is
   * \(\{b^{y_k}\}\) if \(y_k\) has label \(b \in \Sigma\);
   * \([\pi_i(\hat{s}_i)]\) if \(y_k\) is labeled by \(f_i\).
The previous algorithm clearly runs in polynomial time by scanning the tree $T$ and preforming some easy regular language manipulation. Also, the size of $T(\tau_n)$ is linear in the size of the input pair $T$ and $(\tau_n)$. This is clearly true for $R \in \{nFA, dFA\}$ where only a linear number of $\varepsilon$-transitions is required. If $R \in \{nRE, dRE\}$, it is also true because the translation from regular expressions to nFAs produce at most an $n \log^2 n$ blow up but because in these cases we might define $T(\tau_n)$ directly as an nRE-EDTD-type of actual linear size. These considerations immediately yield the following proposition:

**Proposition 3.1.** Given $T(f_n)$ and $(\tau_n)$, the nFA-EDTD-type $T(\tau_n)$ can be constructed in polynomial time, and its size is linear in the input pair.

Now we prove that our construction preserves the semantics of $\text{ext}_T(\tau_n)$.

**Theorem 3.2.** Given a kernel $T(f_n)$ and an $R$-EDTD-typing $(\tau_n)$, $[T(\tau_n)] = \text{ext}_T(\tau_n)$ holds for each possible $R$.

*Proof.* By construction of $T(\tau_n)$, we assume the specialized element names in each type $\tau_i$ of $(\tau_n)$ to be different (in fact, they could always be renamed appropriately before building $T(\tau_n)$). Also, the specialized element names added for giving witnesses to the nodes of $T$ labeled with an element name belong to a fresh set (it is $\tilde{\Sigma}_0$). This means that there is no “competition” among all of these witnesses. So we just create new content models that exactly allow all and only the trees being valid for each $\tau_i$ and the non-function nodes that are already in $T$. But this is exactly the semantic definition of $\text{ext}_T(\tau_n)$. \qed

**Corollary 3.3.** All the problems $\text{CONS}_{\{nFA-EDTD\}}$, $\text{CONS}_{\{dFA-EDTD\}}$, $\text{CONS}_{\{nRE-EDTD\}}$, and $\text{CONS}_{\{dRE-EDTD\}}$ always have a yes answer. Thus, they are decidable in constant time.

*Proof.* For $\text{CONS}_{\{nFA-EDTD\}}$, $\text{CONS}_{\{dFA-EDTD\}}$, and $\text{CONS}_{\{nRE-EDTD\}}$ the decision-answer is always “yes” because each content model in $T(\tau_n)$ is, respectively, already an nFA, expressible by a dFA, and expressible by an nRE.

For $\text{CONS}_{\{dRE-EDTD\}}$ the decision-answer is always “yes” as well, but the reason is less obvious. In general, there are regular languages not expressible by dREs. Anyway, in our case, by considering how $\pi$ is built in $T(\tau_n)$, we are sure that each content model has an equivalent dRE. In fact, $\pi(\tilde{a}_0) = \varepsilon$ (step 4) is already a dRE; $\pi(\tilde{a}_i) = \text{nFA}([\pi_i(\tilde{a}_i)])$ (step 5) has an equivalent dRE because $\pi_i(\tilde{a}_i)$ is already a dRE by definition; $\pi(\tilde{a}_0) = \text{nFA}(L_1 \ldots L_p)$ (step 6) is expressible by a dRE because each $L_k$ originates itself from a dRE and does not share any symbol with any $L_{j \neq k}$. \qed

By Corollary 3.3 we now give a safe and easy construction of $\text{type}_T(\tau_n)$ from $T(\tau_n)$ according to the schema language $S$ used for $(\tau_n)$.

- For nFA-EDTDs, we choose $\text{type}_T(\tau_n) = T(\tau_n)$;
For dFA-EDTDs, we modify \( T(\tau_n) \) by computing the \( \varepsilon \)-closure for each content model. Notice that this can be done in polynomial time and the size of type \( \tau \) is at most quadratic (and there are cases where this could really happen) in the size of \( T(\tau_n) \) because each content model originates from dFAs that do not share any symbol.

For nRE-EDTDs or dRE-EDTDs, we modify the content models of \( T(\tau_n) \) as follows: each \( \pi(\tilde{a}_i) = \pi_i(\tilde{a}_i) \) and each \( \pi(\tilde{a}_0) = R_1 \ldots R_p \) where the generic \( R_k \) is either \( \tilde{b}_0^k \) or \( \pi_i(\tilde{s}_i) \) (compare with the \( T(\tau_n) \) definition). Also here the size of type \( \tau \) is linear in the size of \( T(\tau_n) \) due to the previous corollary.

3.2. \( \mathcal{R} \)-SDTD typing

For \( \mathcal{R} \)-SDTDs we also use \( T(\tau_n) \) as defined for R-EDTDs because any \( \mathcal{R} \)-SDTD can be seen as a special R-EDTD and the algorithm for building \( T(\tau_n) \) still works with no problem. At this point, it should be clear that \( T(\tau_n) \) can easily not be an \( \mathcal{R} \)-SDTD because of our assumptions \( \Sigma_i \cap \Sigma_j \neq \emptyset \). But, in this case it is also possible that \( T(\tau_n) \) does not have an equivalent \( \mathcal{R} \)-SDTD. Indeed, \( T(\tau_n) \) may contain some pattern that already prohibits obtaining an \( \mathcal{R} \)-SDTD for any possible typing \( (\tau_n) \), or it may contain a function-layout that prohibits obtaining an \( \mathcal{R} \)-SDTD for some \( (\tau_n) \). So, we have to discriminate when this is possible or not. Such a problem (deciding whether an \( \mathcal{R} \)-EDTD has an equivalent \( \mathcal{R} \)-SDTD) is in general (when \( \mathcal{R} \) stands for n\( \mathbb{R} \)Es or nFAs) an \textsc{Exptime}-complete problem.

Nevertheless, we will show that, in our case, it is almost always “easier” and in particular that, in general, it depends on the complexity of equivalence between tree-languages specified by nFA-SDTDs, which, in turn depends on the complexity of equivalence between string-languages specified by nFAs. Before giving proofs of that, we illustrate the definition of distributed document using nRE-SDTD-types.

Example 6. Let \( T = s_0(f_1, a(b \ f_2) \ c) \) be a kernel tree and \( \tau_1, \tau_2 \) be two nRE-SDTD-types describing respectively \( b \cdot d^+ \cdot a(b^+)^* \) and \( b^\ast\). In the nRE-SDTDs syntax, \( \tau_1 = \langle \{s_1, a, b, d\}, \{\tilde{s}_1, \tilde{a}_1, \tilde{b}_1, \tilde{d}_1\}, \pi_1, \tilde{s}_1, \mu_1 \rangle \) and \( \tau_2 = \langle \{s_2, b\}, \{\tilde{s}_2, \tilde{b}_2\}, \pi_2, \tilde{s}_2, \mu_2 \rangle \) two types where:

- \( \Sigma_0 = \{s_0, a, b, c\} \); \( \Sigma_0 = \{\tilde{s}_0^3, \tilde{a}_0^3, \tilde{b}_0^3, \tilde{c}_0^3\} \), where \( \{1, 3, 4, 6\} \) are the nodes of \( T \) with label in \( \Sigma_0 \) (based on a preorder traversal of \( T \));
- \( \pi_1(\tilde{s}_1) = \tilde{b}_1 \cdot \tilde{d}_1^+ \cdot \tilde{a}_1^+; \pi_1(\tilde{a}_1) = \tilde{b}_1^+; \pi_1(\tilde{b}_1) = \pi_1(\tilde{d}_1) = \pi_2(\tilde{b}_2) = \varepsilon; \)
- \( \mu_1 \) and \( \mu_2 \) are clear;

For instance, the activation of both \( f_1 \) and \( f_2 \) may return trees \( s_1(bda(bbb)) \) and \( s_2(bb) \), respectively. In general, the resulting type is \( s_0(b \cdot d^+ \cdot a(b^+)^\ast \cdot c) \). It can be described by an nRE-SDTDS. Thus, \( (\tau_1, \tau_2) \) is an nRE-SDTDs-typing consistent with \( T \).

Now we need to introduce some definitions and mention previous results.

Lemma 3.4. Let \( \tau = (\Sigma, \Sigma, \pi, \tilde{s}, \mu) \) be an \( \mathcal{R} \)-SDTD. For each \( \tilde{a} \in \tilde{\Sigma} \), also \( \tau(\tilde{a}) = (\Sigma, \Sigma, \pi, \tilde{a}, \mu) \) is.
Proof. By definition of $R$-SDTD (in the worst case, if $\tau$ is reduced, then $\tau(\tilde{a})$ may be not.)

Definition 15 ([29]). A tree language $L$ is closed under ancestor-guarded subtree exchange if the following holds. For each $t_1, t_2 \in L$, and for each $x_1, x_2$ in $t_1, t_2$, respectively, with $\text{anc-str}_{t_1}(x_1) = \text{anc-str}_{t_2}(x_2)$, the trees obtained by exchanging $\text{tree}_{t_1}(x_1)$ and $\text{tree}_{t_2}(x_2)$ are still in $L$.

Lemma 3.5 ([29]). A tree language is definable by a $R$-SDTD iff it is “closed under ancestor-guarded subtree exchange” and each content model is defined by an $R$-type.

Remark 3. Intuitively, this means that the witness associated by an $R$-SDTD-type $\tau$ to a node $x$ of a tree $t \in [\tau]$ only depends on the string $\text{anc-str}_t(x)$. This is consistent with the definition of $\text{dual}(\tau)$ as a dFA. In fact, the (unique) sequence of states that $\text{dual}(\tau)$ scans for recognizing $\text{anc-str}_t(x)$ (except the initial one) exactly gives the unique witness to each node of $t$ in the path from the root to $x$.


1. There is an equivalent dRE for each one-unambiguous regular language;
2. Let $A$ be a minimum dFA. There is an algorithm, that runs in time quadratic in the size of $A$, deciding whether $[A]$ is one-unambiguous;
3. There are one-unambiguous regular languages where the smallest equivalent dRE is exponential in the size of the minimum equivalent dFA. (This is worst-case optimal);
4. There are one-unambiguous regular languages where some nRE is exponentially more succinct than the smallest equivalent dRE. In particular, the language $\{(a+b)^{m}(a+b)^{n} : m \leq n, n > 0\}$ has such a property;
5. The set of all one-unambiguous regular languages is not closed under concatenation.

Corollary 3.7.

(1) Problem ONE-UNAMB$_{[nRE]}$ is in EXPTIME.
(2) For each nRE defining a one-unambiguous grammar, there exists an equivalent dRE which is, at most, doubly exponential in size. (An exact bound is still open.)
(3) There are pairs of dREs the concatenation of which, by a string separator, defines a one-unambiguous language such that the smallest equivalent dRE has an exponential size.

Proof. (1): Let $r$ be an nRE. Build, in polynomial time from $r$, an equivalent nFA $A$. Run the quadratic-time algorithm described in [11] on the minimum dFA (at most exponentially larger) equivalent to $A$.

(2): By Proposition 3.6 the dRE $r'$ that we construct from the dFA $A$, introduced in (1), has at most size exponential in the size of $A$. Thus, the size of $r'$ is at most doubly exponential in the size of $r$.
(3): Let \( r_1 = (a+b)^m \) and \( r_2 = (a+b)^n \) be to \( \text{nREs} \), with \( m \leq n \). By definition, it is clear that they are also both \( \text{dREs} \) linear in \( n \). Consider the new \( \text{nRE} \) \( r = r_1 r_2 \). By the previous proposition, \( r \) defines a one-unambiguous language but its smallest equivalent \( \text{dRE} \) is exponentially larger. 

Lemma 3.8 \([7]\). Problem \( \text{ONE-UNAMB}_{\text{nRE}} \) is \( \text{PSPACE} \text{-hard} \).

Definition 16. \( \text{CONCAT-UNIV}_{\mathcal{R}} \) is the following decision problem. Let \( \Sigma \) be an alphabet. Given two \( \mathcal{R} \)-types \( \tau_1 \) and \( \tau_2 \) over \( \Sigma \), is \( [\tau_1] \circ [\tau_2] = \Sigma^* \)?

Lemma 3.9 \( [23, 31, 32] \). \( \text{CONCAT-UNIV}_{\mathcal{R}} \) is \( \text{PSPACE} \text{-complete} \) for each \( \mathcal{R} \in \{ \text{nFA}, \text{nRE}, \text{dFA}, \text{dRE} \} \).

After introducing some necessary definitions and results, we are ready for proving the following theorem. It is fundamental for pinpointing the complexity of \( \text{CONS}_{\mathcal{R}-\text{SDTD}} \), for giving size-bounds about \( \text{type}_T(\tau_n) \) and the guidelines for constructing it.

Theorem 3.10. Let \( T(f_n) \) be a kernel and \( (\tau_n) \) be an \( \mathcal{R}-\text{SDTD} \)-typing.

1. If \( \mathcal{R} \in \{ \text{nFA}, \text{nRE} \} \) (nondeterministic and closed under concatenation), then \( \text{CONS}_{\mathcal{R}-\text{SDTD}} \) is polynomial-time Turing reducible to \( \text{EQUIV}_{\mathcal{R}-\text{SDTD}} \) and \( \text{type}_T(\tau_n) \) is not larger than \( T(\tau_n) \);

2. If \( \mathcal{R} = \text{dFA} \) (deterministic and closed under concatenation), then problem \( \text{CONS}_{\mathcal{R}-\text{SDTD}} \) is polynomial-time Turing reducible to \( \text{EQUIV}_{\text{nFA-SDTD}} \) and \( \text{type}_T(\tau_n) \) has unavoidably a single-exponential blow up w.r.t. \( T(\tau_n) \) in the worst case;

3. If \( \mathcal{R} = \text{dRE} \) (not closed under concatenation), then \( \text{CONS}_{\mathcal{R}-\text{SDTD}} \) is polynomial-space Turing reducible to \( \text{ONE-UNAMB}_{\text{nRE}} \). There are cases where the size of \( \text{type}_T(\tau_n) \) is, at least, exponential in the size of \( T(\tau_n) \). A doubly exponential size is sufficient in the worst case. (The exact bound is still open.)

Proof. First of all we observe that, by construction, the only content models of \( T(\tau_n) \) that might not satisfy the single-type requirement are those related to the witnesses of the non-leaf nodes of \( T \). More formally, let \( x \) be any non-leaf node of \( T \) the label of which is denoted by \( a \), the content model \( \pi(\tilde{a}_0^x) \) of its (unique) witness \( \tilde{a}_0^x \) is the only one(s) that may contain some conflict. All other content models either refer to leaves (\( \varepsilon \) is single-type) or come from some \( \tau_i \) (that is already single-type.)

Case 1. Proof Idea: Consider \( (\tau_n) \) being simply an \( \mathcal{R}-\text{EDTD} \). Build \( T(\tau_n) \) and (from it) \( \text{type}_T(\tau_n) = (\Sigma, \tilde{\Sigma}, \pi, s_0, \mu) \) (both in polynomial time as described in Section 3.1) and try to “simplify” the latter (in a bottom-up way starting from the nodes of \( T \) having only leaves as children and going on to the root) for satisfying the single-type requirement. If the algorithm does not fail during its run (\( \text{CONS}_{\mathcal{R}-\text{SDTD}} \) admits a “yes” answer), then the resulting \( \text{type}_T(\tau_n) \) is now an \( \mathcal{R}-\text{SDTD} \). During the proof we only make use of the “ancestor-guarded subtree
exchange” property, and so, by Lemma 3.5, we can conclude that if we cannot simplify \( \text{type}_T(\tau_n) \), then it does not have an equivalent \( \mathcal{R} \)-SDTD. Moreover, due to the simplification process that does not change the structure of \( T(\tau_n) \) but only merges some specialized element names, then the resulting type is at most as large as the original one. Finally, to check the subtree exchange property we only use equivalence between \( \mathcal{R} \)-SDTDs and the number of performed steps is clearly polynomial in the size of \( T(\tau_n) \) witch, by Proposition 3.1, is polynomial in \( T \) and \( (\tau_n) \).

More formally, for each node \( x \) of \( T \) having only leaves as children and of course an element name as label, say \( a \), and for each pair of children \( y \neq z \) of \( x \), do:

1. If both \( y \) and \( z \) are not function nodes and have the same label, say \( b \). As \( \pi(\tilde{b}_0) = \pi(\tilde{b}_0^\prime) = \varepsilon \) (by definition) we can consider hereafter, by Lemma 3.5, \( \tilde{b}_0 \) and \( \tilde{b}_0^\prime \) the same element.

2. If only one of the two, say \( y \), has an element name as label, say \( f_i \), while \( z \) has a function as label, say \( f_j \), and \( \pi_i(\tilde{s}_i) \) contains in its specification an element \( \tilde{b}_i \) (at most one, as \( \tau_i \) is already an \( \mathcal{R} \)-SDTD), by Lemma 3.5 if \( \text{type}_T(\tau_n, \tilde{b}_i) \) is equivalent, then we can consider hereafter, \( \tilde{b}_i \) and \( \tilde{b}_i^\prime \) the same element; otherwise we can conclude that \( \text{type}_T(\tau_n) \) does not have an equivalent \( \mathcal{R} \)-SDTD.

3. Finally, if both \( y \) and \( z \) are function nodes having label \( f_i \) and \( f_j \), respectively, for each element name in \( \Sigma \), say \( b \), if both \( \pi_i(\tilde{s}_i) \) and \( \pi_j(\tilde{s}_j) \) contain in their specifications the elements \( \tilde{b}_i \) and \( \tilde{b}_j \) (at most one for each of them, as \( \tau_i \) and \( \tau_j \) are already \( \mathcal{R} \)-SDTD), by Lemma 3.5 if \( \text{type}_T(\tau_n, \tilde{b}_i) = \text{type}_T(\tau_n, \tilde{b}_j) \) (by construction, this can be done by deciding whether the two \( \mathcal{R} \)-SDTDs \( \tau_i(\tilde{b}_i) \) and \( \tau_j(\tilde{b}_j) \) are equivalent), then we can consider hereafter, \( \tilde{b}_i \) and \( \tilde{b}_j \) the same element; otherwise if for some \( \tilde{b}_i \) and \( \tilde{b}_j \) this is not true, we can conclude that \( \text{type}_T(\tau_n) \) does not have an equivalent \( \mathcal{R} \)-SDTD.

If the corresponding condition is satisfied for each \( y \) and \( z \), then we can conclude that \( \pi(\tilde{a}_0) \) complies with the single-type requirement, that \( \text{type}_T(\tau_n, \tilde{a}_0) \) has an equivalent \( \mathcal{R} \)-SDTD (obtained by applying the previous steps), and that it can be used for checking equivalences when we consider the parent of \( x \), its children and (some modifications of) the three previous steps (see further.)

If \( \text{type}_T(\tau_n, \tilde{a}_0) \) has an equivalent \( \mathcal{R} \)-SDTD for each considered node \( x \), then the next iteration considers each node \( x' \) of \( T \) having only leaves as children or a node already analyzed. We perform Step 3 exactly as above, while Step 1 or Step 2 with the following trivial changes. Let \( y \) be, now, a non-leaf node (instead of a leaf one):

1’. If both \( y \) and \( z \) are not function nodes and have the same label, say \( b \). By Lemma 3.5 if \( \text{type}_T(\tau_n, \tilde{b}_0^\prime) = \text{type}_T(\tau_n, \tilde{b}_0^\prime) \) we can consider hereafter, \( \tilde{b}_0^\prime \) and \( \tilde{b}_0^\prime \) the same element; otherwise we can conclude that \( \text{type}_T(\tau_n) \) does not have an equivalent \( \mathcal{R} \)-SDTD.

2’. If only one of the two, say \( y \), has an element name as label, say \( b \), while \( z \) has a function as label, say \( f_i \), and \( \pi_i(\tilde{s}_i) \) contains in its specification an
Finally, if we reach the root of $T$ and after checking equivalences on its children we can conclude that $\pi(\tilde{s}_0)$ complies with the single-type requirement, then $\text{type}_{\mathcal{T}}(\tau_n, \tilde{b}_0) = \text{type}_{\mathcal{T}}(\tau_n)$ is now (after merging the prescribed specialized element names) a $\mathcal{R}$-SDTD.

Case 2. If $\mathcal{R} = \text{dFA}$ then, when we merge some specialized element names in the same content model, we can obtain an $\text{nFA}$. So, we can still invoke the $\text{equiv}_{\mathcal{R}\text{-SDTD}}$ problem but the size of $\text{type}_{\mathcal{T}}(\tau_n)$ may be exponential as we want it to be a $\text{dFA}$-SDTD, and there are cases for which this may happen already by concatenating two $\text{dFAs}$ [43]. Given that, the blowup cannot be larger than single-exponential, this bound is optimal.

Case 3. If $\mathcal{R} = \text{dRE}$ then, when we merge some specialized element names in the same content model, we can obtain (due to the concatenation and by Proposition 3.6) an $\text{nRE}$ that may not be expressible by a $\text{dRE}$. We can still invoke the $\text{equiv}_{\mathcal{R}\text{-SDTD}}$ problem (as necessary condition) but we also have to invoke the $\text{one-unamb}_{\mathcal{R}\text{-SDTD}}$ problem (at least as hard as the first one). Notice that this new check does not compromise the soundness of the algorithm. In fact, for each possible $\text{dRE}$-$\text{SDTD}$ (if any) equivalent to $\text{type}_{\mathcal{T}}(\tau_n)$ the unique witness that can be assigned to $x$, due to Lemma 3.5 must define the same language as $\pi(\tilde{a}_0^\delta)$ by applying $\mu$ to them. Finally, if both the two decision problems answer yes, then we can consider a new iteration of the previous algorithm. In case that each content model has an equivalent $\text{dRE}$ specification and we reach the root of $T$, we can conclude that $\text{type}_{\mathcal{T}}(\tau_n)$ is now a $\text{dRE}$-SDTD. By Proposition 3.6 there are cases where $\text{type}_{\mathcal{T}}(\tau_n)$ may require, at least, single-exponential size.

We now have the following result:

**Corollary 3.11.**

1. Problems $\text{cons}_{\mathcal{R}\text{-SDTD}}$ and $\text{cons}_{\mathcal{R}\text{-SDTD}}$ are $\text{PSPACE}$-complete;
2. Problem $\text{cons}_{\mathcal{R}\text{-SDTD}}$ is $\text{PSPACE}$-complete;
3. Problem $\text{cons}_{\mathcal{R}\text{-SDTD}}$ is both $\text{PSPACE}$-hard and in $\text{EXPTIME}$;

**Proof.** Membership. For (1) and (2) consider that both $\text{equiv}_{\mathcal{R}\text{-SDTD}}$ and $\text{equiv}_{\mathcal{R}\text{-SDTD}}$ are feasible in $\text{PSPACE}$ [24]. While for (3) we also consider that $\text{one-unamb}_{\mathcal{R}\text{-SDTD}}$ is doable in $\text{EXPTIME}$, by Corollary 3.7.

**Hardness.** For (1) we know that both $\text{equiv}_{\mathcal{R}\text{-SDTD}}$ and $\text{equiv}_{\mathcal{R}\text{-SDTD}}$ are also $\text{PSPACE}$-hard [24].

For (2) and (3) we directly consider a reduction from $\text{concat-univ}_{\mathcal{R}}$ ($\text{PSPACE}$-hard, by Lemma 3.6 to problem $\text{cons}_{\mathcal{R}\text{-SDTD}}$ ($\mathcal{R} \in \{\text{dFA}, \text{dRE}\}$). In particular, let $A_1$, $A_2$ be two $\mathcal{R}$-types, we consider the consistency problem for the kernel tree
\[ T = s(a(f_1, f_2) a(f_3)) \] and the \( R \text{-SDTDs} \) typing \((\tau_1, \tau_2, \tau_3)\) where the trees in \( \tau_1, \tau_2 \) have only one level other than the root, \( \pi_1(s_1) = A_1, \pi_2(s_2) = A_2 \), and \( [\pi_3(s_3)] = \Sigma^* \). It is easy to see that \((\tau_1, \tau_2, \tau_3)\) is consistent with \( T \) if and only if \([A_1] \circ [A_2] = \Sigma^* \).

We conclude this section with a remark.

**Remark 4.** The exponential blow-up affecting \( \text{type}_{\tau}(\tau_n) \) may suggest that there are cases for which it may be better to store an XML document in a distributed manner keeping each part valid w.r.t. its local (and small) type \( \tau_i \) rather than validate the whole document w.r.t. a very large type.

### 3.3. \( R \text{-DTD}s \) typing

Even for \( R \text{-DTD}s \) we use \( T(\tau_n) \) as defined for \( R \text{-EDTDs} \). But here the algorithm we introduced for \( R \text{-EDTDs} \) does not work any more because an \( R \text{-DTD}-typing \) is structurally different from an \( R \text{-SDTD} \) or an \( R \text{-EDTD} \).

Let \( T \) be a kernel and \((\tau_n)\) be an \( R \text{-DTD}-typing \). Before building \( T(\tau_n) \) we construct, from \((\tau_n)\), an equivalent \( R \text{-SDTD}-typing \((\tau'_n)\) as follows. Let \( \tau_i = \langle \Sigma_i, \pi_i, s_i \rangle \) be the \( i \text{-th} \) type in \((\tau_n)\). Consider the \( R \text{-SDTD}-type \( \tau_i = \langle \Sigma_i, \pi, \hat{s}_i, \mu_i \rangle \) defined as follows:

- \( \hat{a} \in \hat{\Sigma}_i \iff a \in \Sigma_i \);
- \( \mu_i \) is a bijection between \( \hat{\Sigma}_i \) and \( \Sigma_i \);
- \( \pi_i'(\hat{a}) = \mu^{-1}(\pi_i(a)) \).

The two types are trivially equivalent. So, we can build the new \( nFA \text{-EDTD-type} \) (or \( nRE \text{-EDTD-type} \), representing \( \text{ext}_T(\tau_n) \), by using \((\tau'_n)\). But since the overhead of constructing \((\tau'_n)\) is completely negligible, we still denote it by \( T(\tau_n) \) instead of \( T(\tau'_n) \).

Also in this case we would like to decide whether \( T(\tau_n) \) has an equivalent \( R \text{-DTD-type} \) or not, and even the general problem (when \( R \) stands for \( nREs \) or \( nFAs \)) of deciding whether an \( R \text{-EDTD} \) has an equivalent \( R \text{-DTD} \) is \text{EXPTIME}-complete [24]. As for \( R \text{-SDTD} \), we will show that in our settings we can do better.

**Definition 17** ([35]). A tree language \( L \) is closed under subtree substitution if the following holds. Whenever for two trees \( t_1, t_2 \in L \) with nodes \( x_1 \) and \( x_2 \), respectively, \( \text{lab}_{t_1}(x_1) = \text{lab}_{t_2}(x_2) \), then the trees obtained, from \( t_1 \) and \( t_2 \), by exchanging \( \text{tree}_{t_1}(x_1) \) and \( \text{tree}_{t_2}(x_2) \) are still in \( L \).

**Lemma 3.12** ([35]). A tree language is definable by an \( R \text{-DTD} \) iff it is “closed under subtree substitution” and each content model is defined by an \( R \text{-type} \).

The following theorem (with the related corollary) concludes the set of results for the bottom-up design problem, and gives the last guidelines for constructing \( \text{type}_{\tau}(\tau_n) \) or evaluating its size.

**Theorem 3.13.** Let \( T(f_n) \) be a kernel and \((\tau_n)\) be an \( R \text{-DTD}-typing \).
1. If $R \in \{nFA, nRE\}$, then $\text{CONS}[R, \text{DTD}]$ is polynomial-time Turing reducible to $\text{EQUIV}[R, \text{SDTD}]$ and $\text{type}_T(\tau_n)$ is linear in $T(\tau_n)$;

2. If $R = dFA$, then $\text{CONS}[R, \text{DTD}]$ is polynomial-time Turing reducible to $\text{EQUIV}[nFA, \text{SDTD}]$ and $\text{type}_T(\tau_n)$ has unavoidably a single-exponential blow up w.r.t. $T(\tau_n)$ in the worst case;

3. If $R = dRE$, then $\text{CONS}[R, \text{DTD}]$ is polynomial-space Turing reducible to $\text{ONE-UNAMB}[nRE]$ and there are cases where $\text{type}_T(\tau_n)$ has at least an exponential blow up w.r.t. $T(\tau_n)$. A doubly exponential size is sufficient in the worst case. (The exact bound is still open.)

Proof. Build the $R$-$\text{SDTD}$-typing $(\tau'_n)$ from $(\tau_n)$ as said before. Perform, from $T$ and $(\tau'_n)$, the decision-algorithm defined in the proof of Theorem 3.10 by enforcing, due to Lemma 3.12, the additional constraint at the end of each macro-step when we assert that $\text{type}_T(\tau_n, \tilde{a}_0)$ has an equivalent $R$-$\text{SDTD}$:

- $[\mu(\pi(\tilde{a}'))] = [\mu(\pi(\tilde{a}''))]$ for each $\tilde{a}'$, $\tilde{a}''$ already considered;

Finally, notice that the (polynomial number) additional steps are special cases of calls to $\text{EQUIV}[R, \text{SDTD}]$, and that the same observations made for $R$-$\text{SDTDs}$ hold for $\text{type}_T(\tau_n)$ as well. 

Corollary 3.14. We have the following results:

1. Problems $\text{CONS}[nRE, \text{DTD}]$ and $\text{CONS}[nFA, \text{DTD}]$ are $\text{PSPACE}$-complete;
2. Problem $\text{CONS}[dFA, \text{DTD}]$ is $\text{PSPACE}$-complete;
3. Problem $\text{CONS}[dRE, \text{DTD}]$ is both $\text{PSPACE}$-hard and in $\text{EXPTIME}$;

Proof. Membership. As for $R$-$\text{SDTDs}$ (see Corollary 3.11).

Hardness. For (1) we know that both $\text{EQUIV}[nRE, \text{DTD}]$ and $\text{EQUIV}[nFA, \text{DTD}]$ are also $\text{PSPACE}$-hard [29, 32].

For (2) and (3) we use the same reduction (from CONCAT-UNIV$[R]$) that we have used in Corollary 3.11 where the problem $\text{CONS}[R, \text{SDTD}]$ is replaced now by $\text{CONS}[R, \text{DTD}]$. In particular, we just notice that also in this case $(\tau_1, \tau_2, \tau_3)$ is consistent with $T = s(a(f_1 f_2) a(f_3))$ if and only if $[\pi_1(\tilde{s}_1) \circ [\pi_2(\tilde{s}_2)] = [\pi_3(\tilde{s}_3)] = \Sigma^*$. 

4. Top-down design

In this section, we consider design problems where we start from a kernel and a given global type, and we show how to reduce each of these problems on trees to a set of typing problems on strings. In the next section, we will show how to solve the problems for strings.
4.1. $\mathcal{R}$-DTDs

We briefly present some obvious results on equivalence of $\mathcal{R}$-DTDs. The proof of the next result is obvious and thus omitted.

**Proposition 4.1.** Two reduced $\mathcal{R}$-DTDs $\tau_1$ and $\tau_2$ are equivalent if and only if the following are true:

1. They have the same root;
2. They use the same element names;
3. For each element name $a$, the content models of $a$ in both are equivalent.

**Theorem 4.2.** Let $D = \langle \tau, T(f_n) \rangle$ be a distributed design where $\tau = \langle \Sigma, \pi, s \rangle$ is an $\mathcal{R}$-DTD. The following are equivalent:

1. $D$ admits a local $\mathcal{R}$-DTD-typing;
2. The $\mathcal{R}$-design $D^x = \langle \pi(\text{lab}(x)), \text{child-str}(x) \rangle$ admits a local $\mathcal{R}$-typing for each node $x$ in $T$ where $\text{lab}(x) \in \Sigma$.

**Proof.** (1) $\Rightarrow$ (2): Let $(\tau_n)$ be a local typing for $D$, then $\text{type}_T(\tau_n) \equiv \tau$ holds. This means (by Proposition 4.1) that for each node $x$ in $T$ such that $\text{lab}(x) \in \Sigma$, the content model $\pi(\text{lab}(x))$ of $x$ has an equivalent specification in $\text{type}_T(\tau_n)$. But this means that the subset of types in $(\tau_n)$ in bijection with the functions of $\text{child-str}(x)$ represents a local typing for $D^x$ as well.

(2) $\Rightarrow$ (1): Also in this case, by Proposition 4.1, since each node $x$ such that $\text{lab}(x) \in \Sigma$ has a local typing, this means that such a typing allows describing exactly the content model $\pi(\text{lab}(x))$. Thus, by combining all the local typings of the various string-designs with the content models of $\tau$ we obtain a $D$-consistent typing also local for $D$. To be more precise, we now show how to exploit the local string-typings for building a local typing for $D$. First of all we observe that, for each $i$ in $[1..n]$, there exists only one node $x$ of $T$ such that $f_i$ is in the kernel string $\text{child-str}(x)$ of $D^x$. Since each $D^x$ admits a local typing, then there is a sequence, say $(\tau_1^{\text{str}}, \ldots, \tau_n^{\text{str}})$, of string-types (one for each function) allowing that. In particular, if for some $x$, $\text{child-str}(x)$ has no function, then this necessarily means that $D^x$ admits a trivial local typing, namely $[\pi(\text{lab}(x))] = \{\text{child-str}(x)\}$ must hold. Let $i$ be an index in $[1..n]$, and $x$ be the parent of $f_i$. The new type (not necessarily reduced) $\tau_i = \langle \Sigma_i, \pi_i, s_i \rangle$ is defined as follows:

- $\Sigma_i = \Sigma \cup \{s_i\}$;
- $\pi_i$ contains all the rules of $\pi$ and the extra rule $\pi_i(s_i) = \tau_i^{\text{str}}$.

Finally, it is very easy to see that, $T(\tau_n)$ is structurally equivalent to $\tau$. $\Box$

**Corollary 4.3.** The problems $\text{LOC|}[\mathcal{R}, \mathcal{DTD}]$, $\text{ML|}[\mathcal{R}, \mathcal{DTD}]$, $\text{PERF|}[\mathcal{R}, \mathcal{DTD}]$, $\exists$-$\text{LOC|}[\mathcal{R}, \mathcal{DTD}]$, $\exists$-$\text{ML|}[\mathcal{R}, \mathcal{DTD}]$, and $\exists$-$\text{PERF|}[\mathcal{R}, \mathcal{DTD}]$ are logspace Turing reducible to $\text{LOC|}[\mathcal{R}]$, $\text{ML|}[\mathcal{R}]$, $\text{PERF|}[\mathcal{R}]$, $\exists$-$\text{LOC|}[\mathcal{R}]$, $\exists$-$\text{ML|}[\mathcal{R}]$ and $\exists$-$\text{PERF|}[\mathcal{R}]$, respectively.
Proof. Let \( D = \langle (\Sigma, \pi, s), T(f_n) \rangle \) be a top-down \( \mathcal{R} \)-SDTD-design.

Consider, firstly, the \( 3\)-LOC[\( \mathcal{R} \)-SDTD] problem. Scan \( T \) in document order, which is well known to be feasible in logarithmic space \([16]\). For each node \( x \) in \( T \) such that \( \text{lab}(x) \in \Sigma \), solve the problem \( 3\)-LOC[\( \mathcal{R} \)] for the design \( D^x \).

If we consider, instead, the problem \( \text{LOC[\( \mathcal{R} \)-SDTD]} \), as \((\tau_n)\) is \( \mathcal{R} \)-consistent, then \( \text{type}_T(\tau_n) \) exists and there are no different content models for the same element name. So it is also enough to scan \( T \) in document order, and for each node \( x \) in \( T \) such that \( \text{lab}(x) \in \Sigma \), solve the problem \( \text{LOC[\( \mathcal{R} \)]} \) for the design \( D^x \) and the subset of types from \((\tau_n)\) in bijection with the functions in \( D^x \).

For the maximal and perfect requirements, as they are specializations of the local requirement, it is enough to observe that, by Theorem 4.5, they only depend on the structure of the various \( D^x \). \( \square \)

4.2. \( \mathcal{R} \)-SDTDs

Before proving that a similar reduction still holds for \( \mathcal{R} \)-SDTDs, we need a proposition and a new definition.

**Proposition 4.4.** Let \( \tau_1 = \langle \Sigma_1, \pi_1, s_1, \mu_1 \rangle \) and \( \tau_2 = \langle \Sigma_2, \pi_2, s_2, \mu_2 \rangle \) be two reduced \( \mathcal{R} \)-SDTDs. If they are equivalent, then for each \( i, j \) in \([1..2]\), and each \( \tilde{a}_i \in \Sigma_i \) there is \( \tilde{a}_j \in \Sigma_j \) such that \( \mu_i(\pi_i(\tilde{a}_i)) = \mu_j(\pi_j(\tilde{a}_j)) \).

**Proof.** As \( \tau_1 \) is reduced, for each specialized element name \( \tilde{a}_1 \in \Sigma_1 \) there is a tree \( t \in [\tau_1] \) such that its unique witness \( t' \) contains at least one node having \( \tilde{a}_1 \) as label. So, let us fix \( \tilde{a}_1, t, \) and a node \( x \) of \( t \) such that \( \text{anc-str}(x) \) ends with the element name \( a \). As \( \tau_1 \) and \( \tau_2 \) are equivalent, there must exist also a unique witness \( t'' \) for \( t \) produced by \( \tau_2 \). Let us denote by \( \tilde{a}_2 \) the specialized element name associated to \( x \) in \( t'' \). As \([\tau_1] = [\tau_2]\), if \( \mu_1(\pi_1(\tilde{a}_1)) \neq \mu_2(\pi_2(\tilde{a}_2)) \), then we would violate the ancestor-guarded subtree exchange property. \( \square \)

**Definition 18.** Let \( D = \langle \tau, T(f_n) \rangle \) be a distributed design where the type \( \tau = \langle \Sigma, \pi, s, \mu \rangle \) is an \( \mathcal{R} \)-SDTD. For each node \( x \) in \( T \) such that \( \text{lab}(x) \in \Sigma \) we denote by \( D^x = \langle \pi(\hat{a}), w_x \rangle \) the the unique string-design induced by \( D \), where \( \hat{a} \) is the (unique) witness assigned by \( \tau \) to \( x \). Moreover, \( w_x = \varepsilon \) if \( x \) is a leaf, and it is the string obtained from \text{children}(x) by changing each non-function node with the corresponding (unique) witness assigned by \( \tau \), otherwise. \( \square \)

**Theorem 4.5.** Let \( D = \langle \tau, T(f_n) \rangle \) be a distributed design where the type \( \tau = \langle \Sigma, \pi, s, \mu \rangle \) is an \( \mathcal{R} \)-SDTD. The following are equivalent:

1. \( D \) admits a local \( \mathcal{R} \)-SDTD-typing;
2. Each \( \mathcal{R} \)-design induced by \( D \) admits a local \( \mathcal{R} \)-typing.

**Proof.** (1) \( \Rightarrow \) (2): Since \( D \) admits a local \( \mathcal{R} \)-SDTD-typing, say \( (\tau_n) \), then \((\tau_n)\) is \( \mathcal{R} \)-SDTD-consistent with \( T \), and \( \text{type}_T(\tau_n) \equiv \tau \) holds. For each node \( x \) of \( T \) having an element name as label, say \( a \), consider the unique witness associated by \( \tau \) and \( \text{type}_T(\tau_n) \) to \( x \), say \( \hat{a}_\tau \) and \( \hat{a}_T \), respectively. By hypothesis, both \( \pi_\tau(\hat{a}_\tau) \) and \( \pi_T(\hat{a}_T) \) satisfy the single-type requirement, and by Proposition 4.4.
Thus, as \( \pi_T(\tilde{a}_x) \) and the children of \( x \) have a local decomposition induced by \( (\tau_n) \), then also \( \pi_T(\tilde{a}_T) \) and the children of \( x \) (namely \( D^x \)) have.

(2) \( \Rightarrow \) (1): We show, if the premise is true, how to build \( (\tau_n) \) in such a way that \( \text{type}_T(\tau_n) \equiv \tau \) holds and in particular that, from a structural point of view, \( \text{type}_T(\tau_n) \) is equivalent to \( \tau \). We denote by \( (\tau^\text{str}_1, \ldots, \tau^\text{str}_n) \) one possible sequence of string-types satisfying contemporarily all the string-designs. In particular, if for some node \( x \) of \( T \) the kernel string of \( D^x \) has no function call this necessarily means that \( D^x \) admits a trivial local typing, namely \( \mu(\pi(\tilde{a})) = \text{child-str}(x) \), where \( \tilde{a} \) is the unique witness assigned by \( \tau \) to \( x \). In particular, for each function \( f_i \) consider its parent node in \( T \), say \( x \). The new type \( \tau_i = \langle \Sigma_i, \tilde{\Sigma}_i, \pi_i, \tilde{s}_i, \mu_i \rangle \) is defined as follows:

- \( \Sigma_i = \Sigma \cup \{ s_i \} \);
- \( \tilde{\Sigma}_i = \tilde{\Sigma} \cup \{ \tilde{s}_i \} \);
- \( \pi_i \) contains all the rules of \( \pi \) and the extra rule \( \pi_i(\tilde{s}_i) = \tau^\text{str}_i \);
- \( \tilde{s}_i \) is the usual extra witness for the root of any tree in \( [\tau_i] \);
- \( \mu_i \) is defined as \( \mu \) and also \( \mu_i(\tilde{s}_i) = s_i \).

It is very easy to see that, if we build \( T(\tau_n) \) without renaming the specialized element names we obtain exactly \( \tau \), and so \( \text{type}_T(\tau_n) \equiv \tau \). In particular, when we assign the witnesses to the non-function nodes of \( T \) we choose exactly those assigned by \( \tau \). The only difference may be in the specification of the content models because the recomposition after a decomposition may produce a different structure (for instance a different \( \text{nFA} \) being, anyway, equivalent to the original one. Notice that, the “ancestor-guarded subtree exchange” property is guaranteed because we also require that all the designs without any function call admit local typings. For instance, if \( D^x = \langle \pi(\tilde{a}), \varepsilon \rangle \) admits a local typing, where \( x \) is a leaf node of \( T \) and \( \tilde{a} \) is its witness assigned by \( \tau \), this necessarily means that \( \pi(\tilde{a}) = \{ \varepsilon \} \), and we automatically take it into account when we build each \( \tau_i \).

\[ \square \]

**Corollary 4.6.** The problems \( \text{LOC}[\mathcal{R}-\text{SDTD}] \), \( \text{ML}[\mathcal{R}-\text{SDTD}] \), \( \text{PERF}[\mathcal{R}-\text{SDTD}] \), \( \exists-\text{LOC}[\mathcal{R}-\text{SDTD}] \), \( \exists-\text{ML}[\mathcal{R}-\text{SDTD}] \) and \( \exists-\text{PERF}[\mathcal{R}-\text{SDTD}] \) are logspace Turing reducible to \( \text{LOC}[\mathcal{R}] \), \( \text{ML}[\mathcal{R}] \), \( \text{PERF}[\mathcal{R}] \), \( \exists-\text{LOC}[\mathcal{R}] \), \( \exists-\text{ML}[\mathcal{R}] \) and \( \exists-\text{PERF}[\mathcal{R}] \), respectively.

**Proof.** Exactly the same as for \( \mathcal{R}-\text{DTDS} \).

\[ \square \]

### 4.3. \( \mathcal{R}-\text{EDTDs} \)

Although \( \mathcal{R}-\text{EDTDs} \) have nice properties simplifying the \( \text{CONS}[\mathcal{R}-\text{EDTD}] \) problem and the construction of \( \text{type}_T(\tau_n) \) (when we start from a kernel and an \( \mathcal{R}-\text{EDTD-} \)typing), things dramatically change when we consider the problems concerning locality. The freedom of using, in the same content model, various specialized element names for the same element name has a price. Consider the following example.
Example 7. Let $D = \langle \tau, T \rangle$ be a dRE-EDTD-design where $T = s_0(f_1 f_2)$ and $\tau = \langle \Sigma, \Sigma, \pi, \pi, s_0, \mu \rangle$. In particular, \( \pi(s_0) = \tilde{a}_1 (\tilde{b}_1)^* + \tilde{a}_2 (\tilde{b}_2)^*; \) \( \pi(\tilde{a}_1) = \tilde{c}_1; \) \( \pi(\tilde{a}_2) = \tilde{d}_1; \) \( \pi(\tilde{b}_1) = \tilde{c}_1 + \tilde{g}_1; \) \( \pi(\tilde{b}_2) = \tilde{g}_1 + \tilde{h}_1. \) It is not hard to see that the string-design $\langle \pi(s_0), f_1 f_2 \rangle$ admits only two maximal local typings:

\[
(\varepsilon, \tilde{a}_1 (\tilde{b}_1)^* + \tilde{a}_2 (\tilde{b}_2)^*) \quad (\tilde{a}_1 (\tilde{b}_1)^* + \tilde{a}_2 (\tilde{b}_2)^*, \varepsilon)
\]

But, only the first one is also maximal for $D$, while the actual second one is $\langle \tilde{a}_1 (\tilde{b}_1)^* + \tilde{a}_2 (\tilde{b}_2)^*, \tilde{b}_3 \rangle$ where $[\tau_2(\tilde{b}_3)] = b(g)$.

The problem highlighted by the previous example originates from the fact that $\tilde{b}_1$ and $\tilde{b}_2$ can not be considered completely distinct as $\tilde{a}_1$ and $\tilde{a}_2$ (notice that $[\tau(\tilde{a}_1)] \cap [\tau(\tilde{a}_2)] = \emptyset$, and as we naturally do for two different symbols of an alphabet in string languages, yet they are witnesses for two sets of trees with a nonempty intersection. In fact, $[\tau(\tilde{b}_1)] \cap [\tau(\tilde{b}_2)] = b(g)$ can be part of $\tau_2$ in the second maximal local typing for $D$.

From this, it is unclear whether, by only analyzing content models (such as $\pi(s_0)$, in the previous example), we can decide whether a given design admits at least a local typing. Clearly, if we apply $\mu$ to both $\langle \tilde{a}_1 (\tilde{b}_1)^* + \tilde{a}_2 (\tilde{b}_2)^* \rangle$ and to $\pi(s_0)$ we obtain the same string-language, namely $ab^*$, but unfortunately, this is only a necessary condition and even if $\langle ab^*, b^* \rangle$ is a maximal local typing for $ab^*$, it is not clear how to assign the witnesses for obtaining $\langle \tilde{a}_1 (\tilde{b}_1)^* + \tilde{a}_2 (\tilde{b}_2)^*, (b^*)^* \rangle$.

The following theorems give a further idea of the higher complexity of locality when we consider $\mathcal{R}$-EDTD-designs.

Theorem 4.7 ([30, 57]). Problems $\text{EQUIV}_{[\mathcal{R}-\text{EDTD}]}$ and $\text{EQUIV}_{[\mathcal{R}, \text{EDTD}]}$ are \textsc{extime}\text{-}complete.

Theorem 4.8. Problems $\exists\text{-}\text{LOC}_{[\mathcal{R}, \text{EDTD}]}$, $\exists\text{-}\text{ML}_{[\mathcal{R}, \text{EDTD}]}$, and $\exists\text{-}\text{PERF}_{[\mathcal{R}, \text{EDTD}]}$ are at least as hard as $\text{EQUIV}_{[\mathcal{R}, \text{EDTD}]}$.

Proof. We define a logspace transformation $\varphi$ from $\text{EQUIV}_{[\mathcal{R}, \text{EDTD}]}$ to $\exists\text{-}\text{LOC}_{[\mathcal{R}, \text{EDTD}]}$. Afterwards, we show that the statement also holds for the other two problems by using exactly the same reduction. Let $\tau', \tau''$ be two arbitrary $\mathcal{R}$-EDTDs. The application of $\varphi$ to this pair produces the design $D = \langle \tau, T \rangle$, where

- $T = s_0(f_1 c f_2)$
- $\pi(s_0) = \mathcal{R}(\tilde{a}_1 \tilde{c}_1 \tilde{d}_1 + \tilde{b}_1 \tilde{c}_1 \tilde{d}_2)$
- $\pi(\tilde{d}_1) = \mathcal{R}(\tilde{s}_0)$, where $\tilde{s}_0$ is the root of the trees in $[\tau']$
- $\pi(\tilde{d}_2) = \mathcal{R}(\tilde{s}_0')$, where $\tilde{s}_0'$ is the root of the trees in $[\tau'']$
- $\pi(\tilde{a}_1) = \pi(\tilde{b}_1) = \pi(\tilde{c}_1) = \mathcal{R}(\varepsilon)$
- $\tilde{c}_1$ does not appear in any other content model of $\tau$ and $c$ appears exactly once in any tree in $[\tau]$

Informally, $|\tau| = s_0(acd(\langle \tau' \rangle) + bcd(\langle \tau'' \rangle))$. First of all, we observe that all the new content models (other than those being already in $\tau'$ and $\tau''$) can be represented by $\mathcal{R}$-types, even $\mathcal{R}$-REs. Now, it is easy to see that $D$ admits a local typing iff $\langle [\tau(\tilde{d}_1)] = [\tau(\tilde{d}_2)] \rangle$ iff $\tau' \equiv \tau''$. It is $[\tau_1] = s_1(a + b)$ and $[\tau_2] = s_2(d(\langle \tau' \rangle))$. Finally, we just notice that if $\tau' \equiv \tau''$ holds, then $(\tau_1, \tau_2)$ is the unique maximal local typing for $D$ which is even perfect. \qed
Corollary 4.9. Problems \(\exists\text{-}\text{LOC}[\mathcal{R}\text{-EDTD}]\), \(\exists\text{-}\text{ML}[\mathcal{R}\text{-EDTD}]\), and \(\exists\text{-}\text{PERF}[\mathcal{R}\text{-EDTD}]\) are \(\text{EXPTIME}\)-hard if \(\mathcal{R} \in \{n\text{FA}, n\text{RE}\}\).

The equivalence between \([\tau(\tilde{d}_1)] = [\tau(\tilde{d}_2)]\), in the previous reduction, is necessary because we do not know, a priori, whether \(f_1\) is imposing a constraint on \(f_2\) or not. In particular, this is an extreme case of the fact that \([\tau(\tilde{d}_1)] \cap [\tau(\tilde{d}_2)] \neq \emptyset\).

What we really need is to be able to consider \textit{completely distinct}, in the same content model, each pair of different specialized element names of the form \(\tilde{a}\) and \(\tilde{a}'\), namely \([\tau(\tilde{a})] \cap [\tau(\tilde{a}')] = \emptyset\). To do that, given an \(\mathcal{R}\text{-EDTD}\), we construct an equivalent \(n\text{UTA}\) \(\tilde{A}\), we transform it into an \(d\text{UTA}\) \(A\), and finally we try to derive a new \(\mathcal{R}\text{-EDTD}\) satisfying our requirement. If \(\mathcal{R} = d\text{RE}\) the last step could not be always possible.

Given an \(\mathcal{R}\text{-EDTD}\) \(\tau = (\Sigma, \tilde{\Sigma}, \pi, s, \mu)\), an equivalent \(n\text{UTA}\) \(A = (K, \Sigma, \Delta, F)\) can be constructed as follows: \(K = \Sigma;\ \Delta(\tilde{a}, a) = n\text{FA}(\pi(\tilde{a}))\), for each \(\tilde{a} \in \tilde{\Sigma}; F = \{s_0\}\). Now we want to transform \(A\) into an equivalent \(d\text{UTA}\) \(A^d\) (that may be exponential in size). Notice that \(A^d\) will have only one final state as well. Finally, we convert again \(A^d\) (whenever it is possible) into an \(\mathcal{R}\text{-EDTD}\) \(\tau^d\) as follows: \(\tilde{\Sigma} = K;\ \pi(\tilde{a}) = R(\Delta(\tilde{a}, a))\), for each \(\tilde{a} \in K\).

Lemma 4.10. Let \(\tau^d\) be an \(\mathcal{R}\text{-EDTD}\) built as above. For each element name, say \(a\), and each pair \(\tilde{a}, \tilde{a}'\) of different specialized element names in \(\tilde{\Sigma}^d(a)\), then \([\tau^d(\tilde{a})] \cap [\tau^d(\tilde{a}')] = \emptyset\).

Proof. It is easy to see that by a (bottom-up) run of \(A^d\) over each tree \(t \in [\tau^d(\tilde{a})] \cup [\tau^d(\tilde{a}')]\), there is only one possible state (between \(\tilde{a}\) and \(\tilde{a}'\)) that can be associated to the root of \(t\), and the states of \(A^d\) coincide with the specialized element names of \(\tau^d\). \(\square\)

Now we are ready for handling \(\mathcal{R}\text{-EDTDs}\) (that we call \textit{normalized}) satisfying the above property. But before we introduce a general property of \(\mathcal{R}\text{-EDTDs}\).

Proposition 4.11. \(\exists\text{-}\text{EDTDs}\). Whenever for two trees \(t_1, t_2 \in [\tau]\) with nodes \(x_1\) and \(x_2\), respectively, there are witnesses \(t_1\) and \(t_2\) assigning the same specialized element name to both \(x_1\) and \(x_2\), then the trees obtained, from \(t_1\) and \(t_2\), by exchanging \(\text{tree}_1(x_1)\) and \(\text{tree}_2(x_2)\) are still in \([\tau]\).

The following lemma holds for general \(\mathcal{R}\text{-EDTDs}\) but is it also useful for normalized \(\mathcal{R}\text{-EDTDs}\). Consider the design \(D = \langle \tau^d, T \rangle\) where \(T = s_0(a(f_1) f_2)\) and \(\tau^d\) is a normalized \(n\text{RE}\text{-EDTD}\) having \(\pi(s_0) = (\tilde{a}_1 + \tilde{a}_2)^+\) (we ignore the other content models). As \([\tau^d(\tilde{a}_1)] \cap [\tau^d(\tilde{a}_2)] = \emptyset\), it is clear that the unique maximal local typing \((\tau_1, \tau_2)\) for \(D\) has \(\pi_1(\tilde{s}_1) = \pi(\tilde{a}_1) + \pi(\tilde{a}_2)\) and \(\pi_2(\tilde{s}_2) = (\tilde{a}_1 + \tilde{a}_2)^*\). Thus the node under the root labeled by \(a\) may have either \(\tilde{a}_1\) or \(\tilde{a}_2\) as witness depending on the tree replacing \(f_1\).

Lemma 4.12. Let \(D = \langle \tau, T \rangle\) be an \(\mathcal{R}\text{-EDTD}\)-design and \((\tau_n)\) be a local typing for \(D\). For each node \(x\) of \(T\) having an element name as label, say \(a\), there is a set of specialized element names \(\tilde{\Sigma}^x \subseteq \tilde{\Sigma}(a)\) such that \(\bigcup_{\tilde{a} \in \tilde{\Sigma}^x} [\tau(\tilde{a})] = [T(\tau_n, \tilde{a}^*)]\).
Proof. By hypothesis, \( [\tau] = \{ T(\tau_n) \} \). We recall that this equivalence is obtained when we consider all the possible extensions of \( T \). Let \( x \) be a node of \( T \) having an element name as label, say \( a \), and \( k \) be the cardinality of \( \Sigma(a) \). First of all, we observe that if \( \Sigma^x = \Sigma(a) \), then

\[
[T(\tau_n, a^0_\sigma)] \subseteq [\tau(\hat{a}^1)] \cup \ldots \cup [\tau(\hat{a}^k)]
\]

is trivially true. In fact, in such a case, the first member has to be necessarily a subset of the second one because, otherwise, in some extension of \( T \) there would be a subtree rooted in \( x \) which \( \tau \) cannot produce any witness for.

Starting from \( \Sigma^x = \Sigma(a) \) we claim that each \( \hat{a}^j \), with \( 1 \leq j \leq k \), is either a “friend” to keep in \( \Sigma^x \) or an “intruder” to remove from \( \Sigma^x \). Finally, the resulting \( \Sigma^x \) will prove the statement. Consider now each \( \hat{a}^j \). We distinguish two cases:

1. We say that \( \hat{a}^j \) is a friend if and only if \( [\tau(\hat{a}^j)] \subseteq [T(\tau_n, a^0_\sigma)] \) because it then clearly contributes to prove the statement. We leave it in \( \Sigma^x \).

2. We say that \( \hat{a}^j \) is an intruder (and we remove it from \( \Sigma^x \)) if and only if one of the following is true:

   * \( [\tau(\hat{a}^j)] \cap [T(\tau_n, a^0_\sigma)] = \emptyset \) because even if we remove it from \( \Sigma^x \), then \( [T(\tau_n, a^0_\sigma)] \subseteq \bigcup_{i \in \Sigma} [\tau(\hat{a}^j)] \) is still true.

   * \( [\tau(\hat{a}^j)] \cap [T(\tau_n, a^0_\sigma)] \neq \emptyset, [\tau(\hat{a}^j)] \not\subseteq [T(\tau_n, a^0_\sigma)] \), and for each (sub)tree \( s \in T(\tau_n) \) in the intersection there is not a tree \( t_1 \in [T(\tau_n)] \) having \( s \) as subtree rooted in \( x \), such that at least a witness \( \tilde{t}_1 \) from \( \tau \) associates \( \hat{a}^j \) to \( x \). It is still an intruder because for each (sub)tree \( s \in T(\tau_n) \) in the intersection and for each possible trees \( t_1 \in [T(\tau_n)] \) having \( s \) as subtree rooted in \( x \), since \( t_1 \) must necessarily have a witness \( \tilde{t}_1 \) from \( \tau \), then \( \tilde{t}_1 \) associates \( \hat{a}^j \) to \( x \) entailing that each \( s \) is contained also in \( [\tau(\hat{a}^j)] \). Finally, even if we remove \( \hat{a}^j \) from \( \Sigma^x \), then \( [T(\tau_n, a^0_\sigma)] \subseteq \bigcup_{i \in \Sigma} [\tau(\hat{a}^j)] \) is still true.

   * \( [\tau(\hat{a}^j)] \cap [T(\tau_n, a^0_\sigma)] \neq \emptyset, [\tau(\hat{a}^j)] \not\subseteq [T(\tau_n, a^0_\sigma)] \), and for some (sub)tree \( s \) in the intersection there is a tree \( t_1 \in [T(\tau_n)] \) having \( s \) as subtree rooted in \( x \) and there is a witness \( \tilde{t}_1 \) from \( \tau \) associating \( \hat{a}^j \) to \( x \). Anyway, this last case is not possible because it would contradict the hypothesis that \( (\tau_n) \) is local. In fact, consider a tree \( t_2 \in \tau \) where some node \( y \) has also witness \( \hat{a}^j \) and the subtree rooted in \( y \) is \( s \in [\tau(\hat{a}^j)] \not\subseteq [T(\tau_n, a^0_\sigma)] \). For each node \( z \) in \( t_1 \) such that \( \text{anc-str}_{\tau_n}(z) = \text{anc-str}_{\tau_n}(x) \) and the witness for \( z \) in \( \tilde{t}_1 \) is \( \hat{a}^j \), by Proposition 4.11, we may replace (in \( t_1 \)) each subtree rooted in \( z \) with \( st_2 \). The new tree is still in \( \tau \) but cannot be obtained by any possible extension of \( T \) as \( [T(\tau_n, a^0_\sigma)] \) contains all the possible trees obtainable by all the extensions of the functions under \( x \) and \( st_2 \) is not among them.

Thus, if we consider all the feasible cases, then the claim is true as well as the theorem: \( \Sigma^x \) is exactly the set of all friends (or a possible subset if some \( [\tau(\hat{a}^j)] \) can be obtained by the union of some \( [\tau(\hat{a}^j)] \) with \( \hat{a}^j \) still in \( \Sigma^x \)). \( \square \)

**Definition 19.** Let \( D = \langle \tau, T(f_n) \rangle \) be an \( R\text{-EDTD} \)-design where the type \( \tau = \langle \Sigma, \Sigma, \pi, s, \mu \rangle \) is normalized. We denote by
• κ any function associating to each node x of T either a set \( \wedge x \subseteq \hat{\wedge}(a) \) if a is the label of x, or the set \( \{ f \} \) if f is the label of x.
• \( D^x = \langle \kappa(x), B^x \rangle \), for each node x in T with \( \text{lab}(x) \in \Sigma \), the box-design induced by D and \( \kappa \) where either \( B^x = \{ \varepsilon \} \) if x is a leaf node, or \( B^x = \kappa(y_1) \ldots \kappa(y_k) \) if \( \text{children}(x) = y_1 \ldots y_k \).

Given a sound typing \((\tau_n)\) for D, we say that
• \( \kappa \) is induced by the pair \((\tau_n)\) and T if, for each non-function node x of T, \( \kappa(x) \) contains exactly all the specialized element names associated to x by validating each possible tree in \( \text{ext}_T(\tau_n) \).
• \( \kappa' \leq \kappa \) iff \( \kappa'(x) \subseteq \kappa(x) \), for each x.

The intention is to relate locality properties about D with locality properties about each \( D^x \) similarly as we made for \( \mathcal{R}\text{-SDTDs} \), with the difference that here \( D^x \) depends on the choice of \( \kappa \). Unfortunately, although \( \tau \) is normalized, if D admits local typings, then \( \kappa \) may not be unique. Consider the following example.

**Example 8.** Let \( D = \langle \tau, T \rangle \) be a normalized \( \mathcal{dRE\text{-}EDTD} \)-design where \( T = s_0(f_1, a(f_2)f_3) \), \( \tau = \langle \Sigma, \hat{\Sigma}, \pi, \hat{s}_0, \mu \rangle \), \( \pi(\hat{s}_0) = \langle \hat{a}^1\hat{a}^2 \rangle^+ \), \( \pi(\hat{a}^1) = \hat{b}^1 \), and \( \pi(\hat{a}^2) = \hat{c}^1 \).

We have two successfully mappings \( \kappa^1, \kappa^2 \) such that
• \( \kappa^1(x_1) = \hat{s}_0, \kappa^1(x_3) = \hat{a}^1, D^x_{\kappa^1} = \langle (\hat{a}^1\hat{a}^2)^+, f_1\hat{a}^1f_3 \rangle \), and \( D^x_{\kappa^2} = \langle \hat{b}^1, f_2 \rangle \)
• \( \kappa^2(x_1) = \hat{s}_0, \kappa^2(x_3) = \hat{a}^2, D^x_{\kappa^1} = \langle (\hat{a}^1\hat{a}^2)^+, f_1\hat{a}^2f_3 \rangle \), and \( D^x_{\kappa^2} = \langle \hat{c}^1, f_2 \rangle \)

From them we have two different maximal local typings for D:
\[
((\hat{a}^1\hat{a}^2)^+, \hat{b}^1, \hat{a}^2(\hat{a}^1\hat{a}^2)^*) \quad ((\hat{a}^1\hat{a}^2)^*\hat{a}^1, \hat{c}^1, (\hat{a}^1\hat{a}^2)^*)
\]
Notice that they are substantially different and also that from the other possible mapping \( \kappa^3 \), where \( \kappa^3(x_3) = \{ \hat{a}^1, \hat{a}^2 \} \), we cannot derive any local typing because if \( f_2 \) is replaced by b, then \( f_3 \) must start with \( a(c) \), and if \( f_2 \) is replaced by c, then \( f_1 \) must start with \( a(b) \). But \( ((\hat{a}^1\hat{a}^2)^*\hat{a}^1, \hat{b}^1 + \hat{c}^1, \hat{a}^2(\hat{a}^1\hat{a}^2)^*) \) is neither local (even) nor sound.

Now we prove the main results of this section.

**Theorem 4.13.** Let \( D = \langle \tau, T(\tau_n) \rangle \) be a distributed design where \( \tau \) is a normalized \( \mathcal{R}\text{-EDTD} \). The following are equivalent:

1. D admits a local typing;
2. There is a function \( \kappa \), as defined above, such that each box-design \( D^x_\kappa \) admits a local typing.

**Proof.**
(1) \( \Rightarrow \) (2): Let \( (\tau_n) \) be a local typing for D, then \( T(\tau_n) \equiv \tau \) holds. Consider the function \( \kappa \) induced by \( (\tau_n) \) and T (the choice is consistent with Lemma 4.12). As \( \tau \) is normalized, there is only one possibility for validating (in a bottom-up way) each tree in \( \text{ext}_T(\tau_n) \). If for some node x of T the box-design \( D^x_\kappa \) did not admit any local typing, then there would be no possibility of generating all the strings in \( \pi(\kappa(x)) \). Contradiction.

(2) \( \Rightarrow \) (1): If for some \( \kappa \) each box-design \( D^x_\kappa \) admits a local typing, then we can construct each type \( \tau_i \) as made for \( \mathcal{R}\text{-SDTDs} \), in such a way that \( T(\tau_n) \) is structurally equivalent to \( \tau \).
Corollary 4.14. Problem $\exists$-LOC$_{[R, ETD]}$ (or $\exists$-ML$_{[R, ETD]}$ but $R \neq dRE$) for normalized $\mathcal{R}$-EDTDs is decidable by an oracle machine in $NP^{C}$ where $C$ is the complexity class of solving $\exists$-LOC$_{[R, ETD]}$ (or $\exists$-ML$_{[R, ETD]}$).

Proof. Let $\tau = \langle \Sigma, \pi, s \rangle$ be a type and $T(!f_n)$ be a kernel. Consider the $\exists$-LOC$_{[R, ETD]}$ problem and the following algorithm:

1. Guess: the function $\kappa$;
2. Check: call $\exists$-LOC$_{[R, ETD]}^B$ over $D_x^\kappa$ for each node $x$ of $T$ with $\text{lab}(x) \in \Sigma$.

For $\exists$-ML$_{[R, ETD]}$ we use the same algorithm since, in general ($R \neq dRE$), a maximal local typing always exists if there is a local one.

Problem $\exists$-ML$_{[RE, ETD]}$ will be discussed in Section 7.

Theorem 4.15. Let $D = \langle \tau, T(f_n) \rangle$ be a distributed design where $\tau$ is a normalized $\mathcal{R}$-EDTD. The following are equivalent:

(1) $D$ admits a perfect typing;
(2) There is a function $\kappa$ such that each $D_x^\kappa$ admits a perfect typing, and for each sound typing $(\tau_n')$ for $D$, $\kappa' \leq \kappa$ where $\kappa'$ is induced by $(\tau_n')$.

Proof. (1) $\Rightarrow$ (2): Let $(\tau_n)$ be the perfect typing of $D$ and $\kappa$ be the function induced by $(\tau_n)$. By Theorem 4.14 each box-design $D_x^\kappa$ admits a local typing, and clearly it is perfect as $(\tau_n)$ is. Finally, we observe that since $(\tau_n') \leq (\tau_n)$, then $(\tau_n')$ can not induce in $\kappa'$ more elements than $(\tau_n)$.

(2) $\Rightarrow$ (1): As we made for $\mathcal{R}$-SDTDs, the typing $(\tau_n)$ that we can construct by the local typings of the various $D_x^\kappa$ (without renaming the specialized element names) together with the needful content models already in $\tau$ produces a type $T(\tau_n)$ structurally equivalent to $\tau$.

Corollary 4.16. Problem $\exists$-PERF$_{[R, ETD]}$ for normalized $\mathcal{R}$-EDTDs is polynomial time reducible to $\exists$-PERF$_{[R, ETD]}^B$.

Proof. Let $\tau = \langle \Sigma, \pi, s \rangle$ be a type and $T(!f_n)$ be a kernel.

Proof Idea: Build $\kappa$ in polynomial time and in a top-down style (this is the technical core of the proof) by assuming that a perfect typing exists. Thus, call $\exists$-PERF$_{[R, ETD]}^B$ over $D_x^\kappa$ for each node $x$ of $T$ with $\text{lab}(x) \in \Sigma$.

More formally, let $x$ be the root of $T$ and $m$ be the number of its children. Consider the following steps:

1. Build from child-str$(x)$ a dRE, that we call $r(x)$, as follows. For each $j$ in $[1..m]$,
   . if child-str$(x)[j]$ is an element name, say $a$, then replace it with the set $\hat{\Sigma}_j(a)$ (where the subscript means that all the specialized element names are renamed with $j$ as subscript);
   . else, if child-str$(x)[j]$ is a function, then replace it with $\hat{\Sigma}_j^*$ ($j$ has the same meaning as above).
2. Build from $\pi(\tilde{s}_0)$ an $R$-type $\tau(x)$ by replacing, in the alphabet of $\pi(\tilde{s}_0)$, each symbol of the form $\tilde{a}^j$ with $\tilde{a}_1^j, \ldots, \tilde{a}_n^j$.
3. Perform the intersection $L = [\tau(x)] \cap [\tau(x)]$.
4. For each child $y$ of $x$ having $a$ as label and position $j$, then $\kappa(y)$ contains all the elements of the form $\tilde{a}$ such that $\tilde{a}_j$ is in the alphabet of $L$.

As we know $\kappa(y)$ for each child $y$ of $x$, then we repeat the previous steps for the children of $y$ by considering $\pi(\kappa(y))$ instead of $\pi(\tilde{s}_0)$. We will stop when we reach the leaves of $T$.

The algorithm is correct because if we have a look at the alphabet of $L$, we see that it contains, for each $j$ in $[1..m]$, exactly the specialized element names that we need to associate to the $j^{th}$ child of $x$ because they are induced by all possible local typings. Intuitively, if the alphabet of $L$ contains, for instance, $\tilde{a}_1^j$ this means that there is a sound typing for $D$ that induces $\tilde{a}^j$ for the first child of $x$, and if the alphabet of $L$ does not contain, for instance, $\tilde{b}_3^j$ there is no sound typing for $D$ inducing $\tilde{b}^j$ for the third child of $x$. $\square$

Now we consider the remaining complexity result that does not require any reduction to strings.

**Theorem 4.17.** Problems $\text{LOC}[R\text{-EDTD}]$, $\text{ML}[R\text{-EDTD}]$, and $\text{PERF}[R\text{-EDTD}]$ are at least as hard as $\text{EQUIV}[R\text{-EDTD}]$.

**Proof.** We define a logspace transformation $\varphi$ from $\text{EQUIV}[R\text{-EDTD}]$ to $\text{LOC}[R\text{-EDTD}]$. Afterwards, we show that the statement also holds for the other two problems. Let $\tau', \tau''$ be two arbitrary $R\text{-EDTD}s$. The application of $\varphi$ to this pair produces the design $D = (\tau, T)$ and the typing $\tau_1$, where $[\tau] = s_0([\tau'])$, $T = s_0(f_1)$, and $[\tau_1] = s_1([\tau''])$. Since $T(\tau_1)$ is exactly $s_0([\tau''])$, it is clear that $\tau \equiv T(\tau_1)$ if and only if $\tau' \equiv \tau''$. Finally, we just notice that $\tau' \equiv \tau''$ iff $\tau_1$ is both perfect and maximal local as $T$ consists of just a function node other than the root. $\square$

**Corollary 4.18.** Problems $\text{LOC}[n\text{FA}\text{-EDTD}]$, $\text{ML}[n\text{FA}\text{-EDTD}]$, and $\text{PERF}[n\text{FA}\text{-EDTD}]$ are $\text{EXPTIME}$-hard.

**Theorem 4.19.** Problem $\text{LOC}[n\text{FA}\text{-EDTD}]$ is $\text{EXPTIME}$-complete.

**Proof.** (Membership) Let $D = (\tau, T)$ be an $n\text{FA}\text{-EDTD}$-design and $(\tau_n)$ be a $D$-consistent typing. Build $T(\tau_n)$ in polynomial time (by Proposition 3.1) and check in exponential time if $T(\tau_n) \equiv \tau$ (by Theorem 4.7).

(Hardness) By Corollary 4.18 $\square$

5. The typing problems for words

We study in this section the typing problems for words. (Recall that most of our problems for trees has been reduced to problems for words.) We present a number of complexity results. We leave for the next section, two issues, namely $\text{PERF}[n\text{FA}]$ and $\exists\text{-PERF}[n\text{FA}]$, for which we will need a rather complicated automata construction. We start by recalling a definition and a result that we will use further.
Theorem 5.1. $\text{EQUIV}_{\text{nFA}}$ is PSPACE-complete.

The hardness of the \text{EQUIV}_{\text{nFA}} problem is used to show some hardness results of our problems.

Theorem 5.2. Problems LOC$_{\text{nFA}}$, ML$_{\text{nFA}}$, PERF$_{\text{nFA}}$ are PSPACE-hard.

Proof. We define a logspace transformation $\phi$, in such a way that

\[ \text{EQUIV}_{\text{nFA}} \leq L \text{ LOC}_{\text{nFA}} \]

Afterwards, we show that the statement also holds for the other two problems. Let $A$, $A_1$ be two arbitrary nFAs. The application of $\phi$ to the pair $A$, $A_1$ produces the design $\langle \tau, w \rangle$ and the typing $\tau_1$, where $\tau = A$, $w = f_1$ and $\tau_1 = A_1$. Since $w(\tau_1) = A_1$, it is clear that $\tau \equiv w(\tau_1)$ if and only if $A \equiv A_1$. Finally, we just notice that $A \equiv A_1$ if and only if $\tau_1$ is both perfect and maximal local as $w$ consists of just a function.

We now consider upper bounds. Section 6 will show that PERF$_{\text{nFA}}$ is in PSPACE. We next show that LOC$_{\text{nFA}}$ is.

Theorem 5.3. LOC$_{\text{nFA}}$ is in PSPACE (so it is PSPACE-complete).

Proof. Let $w(f_n)$ be a kernel string, $\tau$ be an nFA, and $\tau(n)$ be a typing. Since the new automaton $w(\tau_n)$ has size $O(||w|| + |(\tau_n)|)$, we can check in polynomial space if $w(\tau_n) \equiv \tau$.

The proof that also ML$_{\text{nFA}}$ is in PSPACE requires more technical insights and it is deferred to Section 7.

Let us turn to the hardness of the $\exists$-versions of the problems.

Theorem 5.4. $\exists$-LOC$_{\text{nFA}}$, $\exists$-ML$_{\text{nFA}}$, and $\exists$-PERF$_{\text{nFA}}$ are PSPACE-hard.

Proof. We define a logspace transformation $\phi$, in such a way that the following relations hold:

1. $\text{EQUIV}_{\text{nFA}} \leq_m \exists$-LOC$_{\text{nFA}}$;
2. $\text{EQUIV}_{\text{nFA}} \leq_m \exists$-ML$_{\text{nFA}}$;
3. $\text{EQUIV}_{\text{nFA}} \leq_m \exists$-PERF$_{\text{nFA}}$.

Let $A_1 = \langle K_1, \Sigma_1, \Delta_1, s_1, F_1 \rangle$, $A_2 = \langle K_2, \Sigma_2, \Delta_2, s_2, F_2 \rangle$ be two nFAs. The application of $\phi$ to the pair $(A_1, A_2)$ produces the design $D = (A, w)$ where

- $w = f_1 c f_2$, with $c$ being a fresh terminal symbol which does not belong to $(\Sigma_1 \cup \Sigma_2)$;
- while automaton $A = \langle K, \Sigma, \Delta, s, F \rangle$ is defined as follows: (i) $K = K_1 \cup K_2$ $\cup \{s, p_1, q_c\}$; (ii) $\Sigma = \Sigma_1 \cup \Sigma_2 \cup \{a, b, c\}$; (iii) $\Delta = \Delta_1 \cup \Delta_2 \cup \{(s, a, p_c), (s, b, p_c), (p_c, c, q_c), (q_c, e, s_1), (q_c, e, s_2)\}$; (iv) $F = F_1 \cup F_2$. 

38
Intuitively, if we consider \( A_1 \) and \( A_2 \) as nREs, then \( A \) is \((acA_1 + bcA_2)\).

We claim that there is a local typing (similarly, maximal local, or perfect) for \( D \) if and only if \( A_1 \equiv A_2 \). First of all, we observe that transformation \( \varphi \) is extremely simple and it is clearly in logspace. In fact, string \( w \) is a constant, while the choice of a terminal symbol which does not appear in \( A_1 \) nor in \( A_2 \) can be done in logspace, and also \( A \) can be obtained by merging \( A_1 \) and \( A_2 \) with a constant number of transitions. We prove the statement for (1 ) and we just notice that whenever there is a local typing for \( D \), then the typing \(((a + b), A_1)\) is perfect (thus, also maximal).

\((\Rightarrow)\) If there is a local typing for \( D \) then \( A_1 \equiv A_2 \). Since \( A = (acA_1 + bcA_2) \), then \([acA_1]\) and \([bcA_2]\) form a partition of \([A]\). In this case, any local typing must have the following form \(((aX_1 + bX_2), Y)\) where \( X_1, X_2, Y \) are nFA. Clearly, all the strings accepted by \( w \) are obtained by \( aX_1cY \) and \( bX_2cY \). Then \( cA_1 \equiv X_1cY \) and \( cA_2 \equiv X_2cY \) must hold. But since any string in \([A_1]\) or \([A_2]\) does not start with \( c \), then necessarily \([X_1] = [X_2] = \varepsilon\). This way, \( A_1 \equiv Y \) and \( A_2 \equiv Y \) and then \( A_1 \equiv A_2 \).

\((\Leftarrow)\) If \( A_1 \equiv A_2 \), there is a local typing for \( D \). This part of the proof is trivial because \(((a + b), A_1)\) always represents a local typing for \( D \).

We now have lower bounds for all these problems and some upper bounds. We will derive missing upper bounds using the construction of automata that we call “perfect” for given design problems.

6. Perfect automaton for words

We next present the construction of the perfect automaton for a design word problem. The perfect automaton has the property that if a perfect typing exists for this problem, it is “highlighted” by the automaton. This will provide a pSPACE procedure for finding this perfect typing if it exists.

Let \( A = (K, \Sigma, \Delta, s, F) \) be an nFA. We can assume w.l.o.g. that it has no \( \varepsilon \)-transition. Given two states \( q_i, q_f \) in \( K \), a string \( w \) in \( \Sigma^* \) is said to be delimited in \( A \) by \( q_i \) and \( q_f \) if \((q_i, w, q_f) \in \Delta^* \). By exploiting this notion, the sets of all the states delimiting \( w \) in \( A \) are defined as follows:

\[
\text{Ini}(A, w) = \{q_i \in K : \exists q_f \in K \text{ s.t. } (q_i, w, q_f) \in \Delta^*\}
\]

\[
\text{Fin}(A, w) = \{q_f \in K : \exists q_i \in K \text{ s.t. } (q_i, w, q_f) \in \Delta^*\}
\]

In particular, if \( w = \varepsilon \), these two sets are \( \text{Ini}(A, \varepsilon) = \text{Fin}(A, \varepsilon) = K \). \( \text{Ini}(A, w) \) is called the set of initial states while \( \text{Fin}(A, w) \) is the set of final states for the word \( w \). Given two states \( q_i, q_f \) in \( K \), the local automaton \( A(q_i, q_f) = (K' \subseteq K, \Sigma, \Delta', q_i, \{q_f\}) \) induced from \( A \) by \( q_i, q_f \) is a portion of \( A \) containing all those transitions of \( A \) leading from \( q_i \) to \( q_f \). More precisely, for each pair of states \( q, q' \) in \( K \) and for each symbol \( a \) in \( \Sigma \), \((q, a, q') \in \Delta' \) if and only if there are two strings \( u, v \) in \( \Sigma^* \) such that: \((q_i, u, q) \in \Delta^*, (q, a, q') \in \Delta, \) and \((q', v, q_f) \in \Delta^* \). Finally, given two strings \( w_1, w_2 \) in \( \Sigma^+ \), then \( A(w_1, w_2) \) is
the set of all local automata induced by \( w_1 \) and \( w_2 \). It is formally defined as 
\[
A(w_1, w_2) = \{ A(q_i, q_f) : q_i \in \text{Fin}(A, w_1), q_f \in \text{Ini}(A, w_2) \}.
\]
In particular, if \( w_i = \varepsilon \) for some \( i \) in \([1..n]\), the kernel string contains consecutive functions. In particular for the previous definitions we have:

\[
A(w_1, \varepsilon) = \{ A(q_i, q_f) : q_i \in \text{Fin}(A, w_1) \text{ and } q_f \in K \}
\]

\[
A(\varepsilon, w_2) = \{ A(q_i, q_f) : q_i \in K \text{ and } q_f \in \text{Ini}(A, w_2) \}
\]

\[
A(\varepsilon, \varepsilon) = \{ A(q_i, q_f) : q_i, q_f \in K \}
\]

Similarly, given a string \( w \) in \( \Sigma^* \), \( A(w) \) is the set of all local automata induced by \( w \). It is defined as 
\[
A(w) = \{ A(q_i, q_f) : (q_i, w, q_f) \in \Delta^* \}
\]
and in particular \( A(\varepsilon) = \{ A(q, q) : q \in K \} \) is a set of \(|K|\) automata, one for each state in \( K \).

Let \( w(f_n) \) be a kernel string and \( A \) be an nFA. The perfect automaton w.r.t. \( A \) and \( w \) consists of several local automata suitably joined together by \( \varepsilon \)-transitions. It is denoted by \( \Omega(A, w) \) (or \( \Omega \) when it is clear from the context who are \( A \) and \( w \)). Algorithm 1 describes how to build the perfect automaton (assume that any pair of local automata have disjoint sets of states labeled as in \( A \)), while Figure 7 shows the perfect automaton obtained by a given finite state machine and a kernel string. We say that \( A \) is compatible with \( w \) if the set of all (legal) local automata in \( \Omega \) is not empty after correction steps, or equivalently, if there exists at least a sound typing. Moreover,
Algorithm 1 \textsc{PerfectAutomaton}(w, \mathcal{A})

1. \textbf{Input}: $w(f_n) = w_0 f_1 w_1 \ldots f_n w_n$, $\mathcal{A} = (K, \Sigma, \Delta, s, F)$
2. \textbf{Output}: $\Omega(\mathcal{A}, w) = \emptyset$
3. \textbf{for each} automaton $W \in \mathcal{A}(w_0)$ \textbf{do}
    \begin{itemize}
    \item $\triangleright$ add $W$ to $\Omega$
    \end{itemize}
4. \textbf{for each} $i$ in $[1..n]$ \textbf{do}
    \begin{itemize}
    \item $\triangleright$ \textbf{for each} automaton $X \in \mathcal{A}(w_{i-1}, w_i)$ \textbf{do}
        \begin{itemize}
        \item add $X$ to $\Omega$
        \item $\triangleright$ \textbf{for each} automaton $W \in \mathcal{A}(w_i)$ \textbf{do}
            \begin{itemize}
            \item if $\text{label}(q_{\text{fin}}(W)) = \text{label}(q_{\text{ini}}(X))$
                \begin{itemize}
                \item add the transition $(q_{\text{fin}}(W), \varepsilon, q_{\text{ini}}(X))$ to $\Omega$
                \end{itemize}
            \end{itemize}
        \end{itemize}
    \end{itemize}
5. \textbf{for each} automaton $W \in \mathcal{A}(w_0)$ \textbf{do}
    \begin{itemize}
    \item if $\text{label}(q_{\text{ini}}(W)) \neq s$ //if $w_0 = \varepsilon$
        \begin{itemize}
        \item remove $W$ from $\Omega$ //it is illegal
        \end{itemize}
    \end{itemize}
6. merge all automata in $\Omega$ being in $\mathcal{A}(w_0)$ according to their labels and use the (unique) initial state as initial state for $\Omega$
7. \textbf{for each} automaton $W \in \mathcal{A}(w_n)$ \textbf{do}
    \begin{itemize}
    \item if $\text{label}(q_{\text{fin}}(W)) \in F$
        \begin{itemize}
        \item $F(\Omega) = F(\Omega) \cup \{q_{\text{fin}}(W)\}$
        \end{itemize}
    \item else //if $w_n = \varepsilon$
        \begin{itemize}
        \item remove $W$ from $\Omega$ //it is illegal
        \end{itemize}
    \end{itemize}
8. \textbf{for each} automaton $A \in \Omega$ \textbf{do}
    \begin{itemize}
    \item if (there is no path from $q_{\text{ini}}(\Omega)$ to $A$ or
        \begin{itemize}
        \item there is no path from $A$ to any final state of $\Omega$
        \end{itemize}
        \begin{itemize}
        \item remove $A$ from $\Omega$ //it is illegal
        \end{itemize}
    \end{itemize}

- $\text{Seq}(\Omega)$ denotes the set of all the sequences $W_0, X_1, W_1, \ldots, X_n, W_n$ of connected automata in $\Omega$ such that: $W_0$ is an automaton in $\mathcal{A}(w_0)$, while $W_i$ and $X_i$ are, respectively, in $\mathcal{A}(w_i)$ and $\mathcal{A}(w_{i-1}, w_i)$ for any $i$ in $[1..n]$;
- $\text{Typ}(\Omega) = \{(X_i) : W_0, X_1, W_1, \ldots, X_n, W_n \in \text{Seq}(\Omega)\}$ is the set containing all different typings $(X_1, \ldots, X_n)$ from any sequence in $\text{Seq}(\Omega)$;
- $\text{Aut}(\Omega_i) = \{X_i : (X_1, \ldots, X_n) \in \text{Typ}(\Omega)\}$ is the set of all legal automata in $\mathcal{A}(w_{i-1}, w_i)$;
- $\Omega_i = \cup \text{Aut}(\Omega_i)$ is the type obtained by the union of all automata $\text{Aut}(\Omega_i)$;
- $(\Omega_n)$ is the typing for $w$ and $\mathcal{A}$ obtained from $\Omega$.

Let $(\mathcal{A}_n)$ be a sequence of automata. We define the \textit{direct extension} of $(\mathcal{A}_n)$ as the set of string defined as $[(\mathcal{A}_n)] = \{u_1 \ldots u_n \mid \text{for each } i \ u_i \in [\mathcal{A}_i]\}$.  

41
Lemma 6.1. For any nFA \( \mathcal{A} \), then \( \Omega \leq \mathcal{A} \) holds. On the other hand, \( \mathcal{A} \leq \Omega \) does not hold in general.

Proof. Given a string \( u \) in \([\Omega]\), then there exists a sequence \((\tau_{2n+1})\) of automata in Seq(\(\Omega\)) accepting \( u \) and expressible as \( \mathcal{A}(s, q_0), \mathcal{A}(q_0, s_1), \mathcal{A}(s_1, q_1), \ldots, \mathcal{A}(q_{n-1}, s_n), \mathcal{A}(s_n, q_n) \) for some states \( q_0, s_1, q_1, \ldots, s_n, q_n \). Moreover, by definition of direct extension, for each string \( u \sigma_1 u_1 \ldots \sigma_n u_n \) in \([\tau_{2n+1}]\) we have that \( u_0 \in [\mathcal{A}(s, q_0)], \sigma_i \in [\mathcal{A}(q_{i-1}, s_i)] \) and \( u_i \in [\mathcal{A}(s_i, q_i)] \), for each \( i \) in \([1..n]\). But, by definition of local automata, the following sequence of transitions (each of which belongs to \( \Delta^* \)): \((s, w_0, q_0), (q_0, \sigma_1, s_1), (s_1, w_1, q_1), \ldots, (q_{n-1}, \sigma_n, s_n), (s_n, w_n, q_n)\), where \( q_n \in F \), is also derivable by \( \mathcal{A} \).

For the second part of the proof consider the string \( w = abc \) and the dRE \( abc + d \).

Lemma 6.2. Let \( w(f_n) \) be a string compatible with an nFA \( \mathcal{A} \). Any typing in Typ(\(\Omega\)) is sound for \( w \) and \( \mathcal{A} \).

Proof. Given any typing \((X_n)\) in Typ(\(\Omega\)), by definition, there is a sequence \((\tau_{2n+1})\) of automata such that \( X_i = \tau_{2i} \) for each \( i \) in \([1..n]\). By Lemma 6.1 \((\tau_{2n+1}) \leq \mathcal{A} \) holds. Moreover, by definition, the extension of \( w(X_n) \) is \([w(X_n)] = \{w_0 \sigma_1 w_1 \ldots \sigma_n w_n : \sigma_i \in [X_i], 1 \leq i \leq n\}\). Then \( w(X_n) \leq (\tau_{2n+1}) \) as well since all strings \( w_0, \ldots, w_n \) are accepted by \( \tau_1, \tau_3, \ldots, \tau_{2n+1} \), respectively, by definition of local automata induced by a single string. Therefore, \( w(X_n) \leq \mathcal{A} \).

Theorem 6.3. Let \( w(f_n) \) be a kernel string compatible with a given nFA \( \mathcal{A} \), and \((\tau_n)\) be a sound typing for them. Then, both \( w(\tau_n) \leq \Omega \) and \((\tau_n) \leq (\Omega_n) \) hold.

Proof. Since \((\tau_n)\) is sound for \( w \) and \( \mathcal{A} \), then \( w(\tau_n) \leq \mathcal{A} \) holds. In particular, for each string \( \chi = w_0 \sigma_1 w_1 \ldots \sigma_n w_n \) in \([w(\tau_n)]\), where each \( \sigma_i \in [\tau_i] \), there is a sequence of states \( q_0, s_1, q_1, \ldots, s_n, q_n \) proving the membership of \( \chi \) in \([\mathcal{A}]\) by the following sequence of transitions \((s, w_0, q_0), (q_0, \sigma_1, s_1), (s_1, w_1, q_1), \ldots, (q_{n-1}, \sigma_n, s_n), (s_n, w_n, q_n)\) belonging to \( \mathcal{A} \) as well. But, this means that the sequence \( \mathcal{A}(s, q_0), \mathcal{A}(q_0, s_1), \mathcal{A}(s_1, q_1), \ldots, \mathcal{A}(q_{n-1}, s_n), \mathcal{A}(s_n, q_n) \) of automata belongs to Seq(\(\Omega\)), so \( w(\tau_n) \leq \Omega \) holds.

Moreover, since each \( \mathcal{A}(q_{i-1}, s_i) \in Aut(\Omega_i) \), it follows that \( \tau_i \leq \Omega_i \) for each \( i \), that is \((\tau_n) \leq (\Omega_n) \).

Corollary 6.4. Let \( w(f_n) \) be a kernel string compatible with a given nFA \( \mathcal{A} \), and \((\tau_n)\) be a local typing for them. Then, \( w(\tau_n) \equiv \Omega \equiv \mathcal{A} \) holds.

Proof. By Lemma 6.1 and Theorem 6.3.

Theorem 6.5. Let \( w(f_n) \) be a kernel string and \( \mathcal{A} \) be an nFA compatible with \( w \). There is a perfect typing for \( w \) and \( \mathcal{A} \) if and only if \( w(\Omega_n) \equiv \mathcal{A} \). If so, the perfect typing is exactly \((\Omega_n) \).
Proof. ($\Rightarrow$) if there is a perfect typing for $w$ and $A$ then $w(\Omega_n) \equiv A$. If $w$ and $A$ admit a perfect typing, say $(\tau_n)$, then (as it is also sound), by Theorem 6.3 $(\tau_n) \leq (\Omega_n)$. Suppose that $(\tau_n) < (\Omega_n)$ held. There would be (at least) an $i$ in $[1..n]$ such that $\tau_i < \Omega_i$. In other words, there would be an automaton $\tau'_i \in \text{Aut}(\Omega_i)$ accepting some strings rejected by $\tau_i$. Consider the typing $(\tau'_i) \in \text{Typ}(\Omega)$ containing $\tau'_i$ in position $i$. By Lemma 6.2 $(\tau'_i)$ is sound and then $\tau'_i \leq \tau_i$, by definition. But this is a contradiction. Therefore $(\tau_n) \equiv (\Omega_n)$ and then $w(\Omega_n) \equiv A$, as $(\tau_n)$ is also local.

($\Leftarrow$) if $w(\Omega_n) \equiv A$ then there is a perfect typing for $w$ and $A$. This is true since $(\Omega_n)$ is local and because, by Theorem 6.3 $(\tau_n) \leq (\Omega_n)$ for any sound typing $(\tau_n)$.

The following two examples show that if there exists a local typing $(\tau_n)$ for $w$ and $A$, then $(\tau_n) < (\Omega_n)$ might hold. This can happen even if $(\tau_n)$ is a unique maximal local.

Example 9. Consider the string $w = a f_1 c f_2 e$, and the regular expression $\tau = abcced$ compatible with $w$. Clearly, the typing $(b, cd)$ is local (sound and complete) for $w$ and $\tau$ because $w(b, cd) \equiv \tau$. Nevertheless, $(\Omega_2) = (bc\:? , c?d)$ is (strictly) greater then $(b, cd)$ since $[bc\:?] = \{b, bc\} \supset \{b\}$ and $[c?d] = \{d, cd\} \supset \{cd\}$.

Example 10. Let $w = a f_1 f_2 d$ be a kernel string and $\tau$ be the regular expression $(bc)^*d$. Clearly, the typing $((bc)^*, (bc)^*)$ is local (also unique maximal local but not perfect). But, as consequence of the construction of perfect automaton, we have: $\text{Aut}(\Omega_1) = \{(bc)^*, (bc)^*b\}$ and $\text{Aut}(\Omega_2) = \{(bc)^*, c(bc)^*\}$. Consequently, $\Omega_1 \equiv ((bc)^*b\:?)$ and $\Omega_2 \equiv (c?bc)^*$ do not represent a sound (and hence local) typing since they allow strings such as $aeebcd$ or $abecbd$ that are not accepted by $\tau$.

The following example shows that even if there is no local typing for $w$ and $\tau$, then $\Omega \equiv \tau$ may hold.

Example 11. Let $\tau$ be the regular expression $ab + ba$ and $w = f_1 f_2$. There are two sound typings: $(a, b)$ and $(b, a)$, but there is no local typing. However, $\Omega \equiv \tau$.

We can now use the perfect automata construction to characterize the complexity of $\text{PERF}[\text{naFA}]$. We use the next lemma:

**Lemma 6.6.** Let $w(f_n)$ be a kernel string and $A$ be a $k$-state $\text{nFA}$. The algorithm for building the perfect automaton $\Omega(A, w)$ works in polynomial time.

**Proof.** Any set $A(w_1)$ or $A(w_{i-1}, w_i)$ contains at most $k^2$ automata each of which having size $O(k)$. Therefore, the number of macro-iterations of the algorithm are $O(nk^2)$, while the size of $\Omega$ is $O(nk^3)$. For each $w_i$, the sets $\text{Init}(A, w_i)$ and $\text{Fin}(A, w_i)$ can be obtained in nondeterministic logarithmic space (thus in polynomial time) because for any pair of states $q_1, q_2$ in $A$, we check if the
string $w_i$ is in the language $[A(q_1, q_2)]$. Finally, all the automata in $A(w_i)$ and $A(w_i, q_i)$ are nothing else but different copies of $A$ having different initial and final states.

Now, we have:

**Theorem 6.7.** \(\text{PERF}_{n\mathcal{FA}}\) is in \(PSPACE\). So it is also \(PSPACE\)-complete by Theorem 5.2

**Proof.** Let $w(f_n)$ be a kernel string, $\tau$ be an $n\mathcal{FA}$, and $(\tau_n)$ be a typing. Construct the perfect automaton $\Omega(\tau, w)$. By Lemma 6.6, $\Omega$ can be built in polynomial time w.r.t. $|\tau| + \|w\|$. Then, check in polynomial space if $w(\Omega_n) \equiv \tau \equiv w(\tau_n)$. \hfill \square

And w.r.t. finding a perfect typing (if it exists), we have:

**Theorem 6.8.** $\exists$-\(\text{PERF}_{n\mathcal{FA}}\) is in \(PSPACE\). So it is also \(PSPACE\)-complete by Theorem 5.4

**Proof.** Let $(\tau, w(f_n))$ be a (string) design. Construct the perfect automaton $\Omega(\tau, w)$. By Lemma 6.6, $\Omega$ can be built in polynomial time w.r.t. $|\tau| + \|w\|$. Then, check if $w(\Omega_n) \equiv \tau$, which is feasible in polynomial space. \hfill \square

### 6.1. Additional properties

We now show how to exploit perfect automaton properties to find (maximal) sound typings when a design does not allow any perfect. Clearly, this technique can be used for seeking (maximal) local typings as well. Let $w(f_n)$ be a kernel string and $A$ be an $n\mathcal{FA}$-type compatible with $w$. All the automata belonging to $\text{Aut}(\Omega_i)$ can be decomposed in at most $2^{\text{Aut}(\Omega_i)} - 1$ different automata such that there are no two of them accepting the same string. In particular, this new set is denoted by $\text{Dec}(\Omega_i)$ and defined as follows:

$$\text{Dec}(\Omega_i) = \{\cap A_1 - \cup A_2 : \emptyset \neq A_1 \subseteq A(\Omega_i), A_2 = A(\Omega_i) - A_1\}$$

An example for three automata is given in Figure 3. Finally, $\text{Dec}(\Omega) = \{(D_1, \ldots, D_n) : D_i \in \text{Dec}(\Omega_i)\}$ is the set of all different typings from $\text{Dec}(\Omega_1) \times \ldots \times \text{Dec}(\Omega_n)$.

Given a typing $(\tau_n)$, we say that $(\tau_n) \in \text{Dec}(\Omega)$ if there exists a sequence $(D_n) \in \text{Dec}(\Omega)$ such that $\tau_i \equiv D_i$, for each $i$.

Given a type $\tau \subseteq \Omega_i$ for some $i$ in $[1..n]$, $\text{Dec}(\tau, i) = \{\tau \cap \tau' : \tau' \in \text{Dec}(\Omega_i)\}$ denotes the partition of $\tau$, namely $\cup \text{Dec}(\tau, i) = \tau$, obtained by its projection on $\text{Dec}(\Omega_i)$. Let $(\tau_n)$ be any typing for a kernel string $w(f_n)$. Given a string $u \in \Sigma^*$ and an $i$ in $[1..n]$, then $(\tau_n)[\tau_n, u]$ denotes the new typing obtained from $(\tau_n)$ by replacing $\tau_i$ with the minimal $d\mathcal{FA}$ accepting only the string $u$. In particular $[w(\tau_n)[\tau_n, u]]$ is defined as $\{w_i \sigma_1 w_1 \ldots \sigma_n w_n : \sigma_i = u, \sigma_j \in [\tau_j] \forall j \neq i\}$ and clearly,

$$w(\tau_n) \equiv \bigcup_{u \in [\tau_i]} w(\tau_n)[\tau_n, u]$$
We now define an extension of \((\tau_n)\) as the new typing obtained from \((\tau_n)\) by replacing \(\tau_i\) with the new type \((\tau_i \cup \tau)\), and denoted by \((\tau_n)_{|\tau_i \cup \tau}\). In particular,

\[
w(\tau_n)_{|\tau_i \cup \tau} = \bigcup_{\omega \in [\tau_i \cup \tau]} w(\tau_n)_{|\tau_i, u}
\]

Clearly, if \(\tau \leq \tau_i\), then \((\tau_n) \equiv (\tau_n)_{|\tau_i \cup \tau}\). Otherwise \((\tau_n) \neq (\tau_n)_{|\tau_i \cup \tau}\).

**Definition 20.** A type \(\tau\) extends another type \(\tau'\) if \([\tau] - [\tau'] \neq \emptyset\) holds. Moreover, the extension is called **partial** or **total** depending on whether \([\tau] \cap [\tau'] \neq \emptyset\) or not, respectively.

**Lemma 6.9.** Let \(D = (A, w)\) be an \(n\text{FA}\)-design, \((\tau_n)\) be a consistent sound typing for \(D\), and \(\tau \in \text{Dec}(\Omega_i)\) be an \(n\text{FA}\) belonging to the decomposition of \(\Omega_i\), for some \(i \in [1..n]\). If \(\tau\) partially extends \(\tau_i\), then the extension \((\tau_n)_{|\tau_i \cup \tau}\) of \((\tau_n)\) is still sound.

**Proof.** By definition 20 \([\tau]\) contains at least a string that does not belong to \([\tau_i]\) but also a string, say \(u'\), accepted by both \(\tau_i\) and \(\tau\). In order to prove the statement, we show that \(w(\tau_n)_{|\tau_i, u} \leq \Omega\) holds for each \(u \in [\tau] - [\tau_i]\) (recall that, by Lemma 6.1, \(\Omega \leq A\)).

Since, by Theorem 6.3 \(\tau_i \leq \Omega_i\), then there is a nonempty set \(A \subseteq \text{Aut}(\Omega_i)\) containing all-and-only the automata accepting \(u'\). Clearly, since \(\tau \in \text{Dec}(\Omega_i)\) and \(u' \in [\tau]\), then \(\tau\) is also in \(\text{Dec}(\tau', i)\) for each \(\tau' \in A\). This means that each string \(u \in [\tau] - [\tau_i]\) is accepted by all-and-only the automata in \(A\) as well. By Theorem 6.3 \(w(\tau_n) \leq \Omega\), and in particular \(w(\tau_n)_{|\tau_i, u'} \leq \Omega\), as \(u' \in [\tau_i \cap \tau]\). In other words, any string in \([w(\tau_n)_{|\tau_i, u'}]\) is accepted by (at least) a sequence of automata in \(\text{Seq}(\Omega)\). Finally, as both \(u\) and \(u'\) are recognized by all-and-only the automata in \(A\), then each string \(w_0 \sigma_1 \ldots \sigma_n w_n\) in \([w(\tau_n)_{|\tau_i, u'}]\) (with \(\sigma_i = u\)) has a twin in \([w(\tau_n)_{|\tau_i, u'}]\) (with \(\sigma_i = u'\)) and both of them are accepted by exactly the same sequences in \(\text{Seq}(\Omega)\).

**Theorem 6.10.** Let \((\tau_n)\) be a maximal typing for a kernel string \(w(f_n)\) and an \(n\text{FA}\) \(A\) compatible with \(w\). Then for each \(i\), \(\text{Dec}(\tau_i, i) \subseteq \text{Dec}(\Omega_i)\).
Proof. Let i be an index arbitrarily fixed in [1..n]. As (τ_n) is maximal then, by definition, it is sound and, by Theorem 6.3, τ_i ≤ Ω_i. Let D_i be a copy of Dec(Ω_i). Then τ_i ⊆ ∪D_i. Remove now, from D_i, each automaton τ_{D_i} (if any) such that [τ_{D_i}] ∩ [τ_i] = ∅. Still, τ_i ⊆ ∪D_i holds. Hence, consider the two possible (and alternative) cases: (1) τ_i ≡ ∪D_i, or (2) τ_i < ∪D_i. In the first case the theorem is already proved. While, in the latter case, there is (at least) an automaton τ ∈ D_i that partially extends τ_i entailing relation (τ_n) < (τ_n)[τ_i∪τ]. But since (τ_n)[τ_i∪τ] is still sound (see Lemma 6.9), then there is a contradiction because (τ_n) is assumed to be maximal. □

We are now ready to prove a main results of the section. In our original paper, we showed a 2-EXPSPACE upper bound for ∃-LOC[∩FA] and ∃-ML[∩FA]. This was improved to EXPSPACE in [31]. We present here an alternative proof of that results using the previous decomposition.

**Theorem 6.11.** Problems ∃-LOC[∩FA] and ∃-ML[∩FA] are in EXPSPACE.

**Proof.** By Lemma 6.9 and Theorem 6.10, if an nFA-design D = ⟨τ, w⟩ admits a (maximal) local typing, say (τ_n), then for each τ_i there exists a subset of Dec(Ω_i), say D_i, such that ∪D_i ≡ τ_i.

Let m be the number of states of τ, and ν + n be the length of w where n is clearly the number of functions and ν is the length of the non-function symbols in w. By definition, for each i in [1..n], each automaton in Aut(Ω_i) has size at most m and the cardinality of Aut(Ω_i) is at most m^2. Thus, the cardinality of Dec(Ω_i) is no more than 2^{m^2}, as well as the cardinality of D_i. In the worst case, an automaton in Dec(Ω_i) is obtained as ∩A_1 ∩ A_2 where both |A_1| = |A_2| = O(m^2). So, the size of ∩A_1 is no more than (m^2)^m^2 [22], that is clearly lower than 2^{m^2}. The size of ∪A_2 is at most m^3 [22]. Now, for computing ∩A_1 ∩ A_2 we perform the following intersection (∩A_1) ∩ (∪A_2). The complement of ∪A_2 may have 2^{m^3} states [22]. Finally ∩A_1 ∩ A_2 require no more than 2^{m^3} states, and the size of ∪D_i ≡ τ_i is at most 2^{2m^3} * 2^{m^2} being clearly 2^{O(m^3)}.

Now, we are ready for computing the size of the nFA w(τ_n). It is exactly ν + n + 2^{O(m^3)}. So, for deciding whether w(τ_n) ≡ τ we need no more than exponential space w.r.t. the input size ν + n + m. The only problem we still have is that we do not know a priori how to choose D_i. There are 2^{2m^3} possible subsets. But as NEXPSPACE = EXPSPACE (by Savitch’s theorem), then we can simply guess each D_i.

About ∃-ML[∩FA], we must find a maximal D_i. But in EXPSPACE we can still guess the sequence D_1, . . . , D_n and prove (by Theorem 6.10), for each D_i, that none of the automata in Dec(Ω_i) − D_i can be added to D_i because the resulting typing would loose its soundness. After the guess, the number of checks (each of which may require exponential space) is at most n * 2^{m^2}. □
7. Complexity for trees

Based on Theorem 6.11 we now obtain complexity bounds for the tree problems. This completes results obtained in [3, 31] on this topic. The next result first appeared in [3]. However, the sketch of proof given there was not correct. A proof was then presented in [31]. We next present a new proof based on perfect automata.

**Theorem 7.1.** \( M_{\lfloor nFA \rfloor} \) is in \( PSPACE \) (so the problem is \( PSPACE \)-complete).

**Proof.** Let \( D = (\tau, w) \) be an nFA-design, and \( (\tau_n) \) be a \( D \)-consistent typing. First of all, we check if \( (\tau_n) \) is local (and we have already proved that \( loc_{\lfloor nFA \rfloor} \) is doable in \( PSPACE \)). If so, then \( \bar{\tau} \cap w(\tau_n) \equiv \emptyset \) (where \( \bar{\tau} \) is the nFA of possibly exponential size accepting the complement of language \( [\tau] \)). Subsequently, we check if \( (\tau_n) \) is not maximal. In particular, by Lemma 6.9 and Theorem 6.10, \( (\tau_n) \) is not maximal if there is an nFA \( A \in Dec(\Omega_i) \) for some \( i \in [1..n] \) such that at least one of the following is true:

- \( A \) totally extends \( \tau_i \) and \( w(\tau_n) |_{[\tau \cup A]} \) is still sound, namely \( [A] \cap [\tau_i] = \emptyset \) and \( \bar{\tau} \cap w(\tau_n) |_{[\tau \cup A]} \equiv \emptyset \).
- \( A \) partially extends \( \tau_i \), namely \( [A] \cap [\tau_i] \neq \emptyset \) and \( [A] \cap [\tau_i] \neq \emptyset \).

So we proceed as follows:

1. Guess an index \( i \), and a nonempty set of automata \( A_1 \subseteq Aut(\Omega_i) \)
2. Compute \( A_2 = Aut(\Omega_i) - A_1 \)
3. Let \( A \) denote the automaton \( \cap A_1 - \cup A_2 \) (we do not really build it);
4. If \( [A] \cap [\tau_i] = \emptyset \) then,
   - if \( \bar{\tau} \cap w(\tau_n) |_{[\tau \cup A]} \equiv \emptyset \), then \( (\tau_n) \) is not maximal
   - else if \( [A] - [\tau_i] \neq \emptyset \) and \( [A] \cap [\tau_i] \neq \emptyset \), then \( (\tau_n) \) is not maximal

Observe that even if \( A \), \( \bar{\tau} \), or \( \bar{\tau}_i \) may be exponential in size, we only use them for intersection nonemptiness or intersection emptiness problems that are both \( NL \)-complete problems [20]. Intuitively, we could avoid the materialization of such automata with “on-the-fly” constructions. Hence, an \( NL \) algorithm on a non-materialized (single) exponential automaton leads to \( PSPACE \). More formally, we consider alternating finite state machines \( aFAs \) (for more details see [17, 42]). We do not completely define them but we just recall what we need:

- given an \( aFA \) \( A \), deciding whether \( [A] = \emptyset \) is \( PSPACE \)-complete;
- Any nFA is trivially a special kind of aFA;
- Given two aFAs \( A \) and \( A' \), a new aFA for \( \overline{A' \cup A} \), and \( A \cap A' \), can be constructed in polynomial time and its size is linear.

Finally, we observe that all the above emptiness decisions deal with nFAs of polynomial size and can be checked in \( PSPACE \) as well as the nonemptiness decisions as \( PSPACE \) is closed under complement.

Now, we show how to reduce locality problems on boxes to locality problems on strings.
Definition 21. Let $D = \langle \tau, B \rangle$ be an $R$-design where $B = B_0B_1 \ldots B_n$ is a kernel box. Consider the $k^{th}$ sequence of strings $w_0, \ldots, w_n$ (with $1 \leq k \leq |B_0| \ldots |B_n|$) built from $B_0, \ldots, B_n$ by varying $w_i$ among the strings in $[B_i]$ (in some fixed order) for each $i \in [0..n]$. We denote by $D^k = \langle \tau, w^k \rangle$ the $k^{th}$ $R$-design built from $D$ where $w^k(f_n)$ is the kernel string $w_0f_1w_1 \ldots f_nw_n$.

Lemma 7.2. Let $D = \langle \tau, B \rangle$ be an $R$-design and $(\tau_n)$ be a $D$-consistent typing. We have that:

1. If $(\tau_n)$ is local for $D$, then it is sound for each $D^k$;
2. If $(\tau_n)$ is sound for each $D^k$, then it is sound for $D$ as well.

Proof. (1): If $(\tau_n)$ is local for $D$, then $B_0\tau_1B_1 \ldots \tau_nB_n \equiv \tau$. This means that $w_0\tau_1w_1 \ldots \tau_nw_n \leq \tau$ for each $w_i \in [B_i]$. Thus, $(\tau_n)$ is sound for each $D^k$.

(2): If for each design $D^k$ we have that $w_0\tau_1w_1 \ldots \tau_nw_n \leq \tau$ holds, then $B_0\tau_1B_1 \ldots \tau_nB_n \leq \tau$ as well.

A direct consequence of the above theorem is that if a typing is not sound for some $D^k$, then it can not be local for $D$. So a local typing candidate for $D$ is a typing being sound for each $D^k$. Now suppose that $(\tau_n)$ is a maximal sound typing for $D^k$ but it is not sound for $D^{k^2}$. This means that at least one $[\tau_i]$ contains some extra string such that $[w^{k^2}(\tau_n)]$ is not fully contained in $[\tau]$. So we could remove such strings to obtain a typing sound for both $D^k$ and $D^{k^2}$ but not maximal for $D^{k^2}$ any more. So we can guess a maximal sound typing for each $D^k$ and then, remove the exceeding strings. This is equivalent to keeping the componentwise intersection of these maximal typings. Let $\beta = [B_0] \ldots [B_n]$, we should build $\beta$ (it is an exponential number) perfect automata. For each $i$ in $[1..n]$ we should consider the sets of automata $Aut(\Omega_1^i), \ldots, Aut(\Omega_n^i)$ and from these the respective decompositions $Dec(\Omega_1^i), \ldots, Dec(\Omega_n^i)$. Now we can guess $\beta$ subsets $D_1^i, \ldots, D_n^i$ and finally compute $\tau_i$ as $(\cup D_1^i) \cap \ldots \cap (\cup D_n^i)$. But this is equivalent to consider directly $Aut(\Omega_i) = Aut(\Omega_1^i) \cup \ldots \cup Aut(\Omega_n^i)$, compute the decomposition $Dec(\Omega_i)$ and guess a subset $D_i$ from $Dec(\Omega_i)$. This is much more convenient because $Aut(\Omega_i)$ contains at most a quadratic number of automata w.r.t. the states of $\tau$. Now, we show how to extend the construction of $\Omega$ to a box-design for obtaining this new $Aut(\Omega_i)$. Let $\mathcal{A}$ be an nFA and $B(f_n)$ a kernel box, we have that:

- $Ini(\mathcal{A}, B_i) = \{q_i \in K : \exists q_f \in K \text{ s.t. } (q_i, w, q_f) \in \Delta^*, w \in [B_i]\}$
- $Fin(\mathcal{A}, B_i) = \{q_f \in K : \exists q_i \in K \text{ s.t. } (q_i, w, q_f) \in \Delta^*, w \in [B_i]\}$
- $A(B_{i-1}, B_i) = \{A(q_i, q_f) : q_i \in Fin(\mathcal{A}, B_{i-1}), q_f \in Ini(\mathcal{A}, B_i)\}$

$Aut(\Omega_i)$ is the set of all legal automata in $A(B_{i-1}, B_i)$ as for string. Note that, due to the structure of each $B_i$, it is very easy to build $Ini(\mathcal{A}, B_i)$ and $Fin(\mathcal{A}, B_i)$ without enumerating all the strings in $[B_i]$. 

Theorem 7.3. Problems $\exists$-LOC$^B_{nFA}$ and $\exists$-ML$^B_{nFA}$ are in EXPSPACE.
Proof. Let $D = \langle \tau, B \rangle$ be an $nFA$-design where $B$ is a kernel box. We guess, for each $i$, a subset of automata in $Dec(\Omega_i)$, the decomposition of the new set $Aut(\Omega_i)$ built as above. Thus, we check if it is a (maximal) local typing for $D$ as made in the proof of Theorem 7.4. □

**Corollary 7.4.** $\exists:\text{LOC}_{[\text{nFA-EDTD}]}^\beta$ and $\exists:\text{ML}_{[\text{nFA-EDTD}]}^\beta$ are in $2\text{-EXPSPACE}$.

Proof. Let $D = \langle \tau, T \rangle$ be an $nFA$-EDTD-design. We build from $\tau$ its equivalent normalized version $\tau^d$ that, after all, is a $dFA$-EDTD of exponential size. So the oracle machine discussed in Corollary 4.14 actually works in $\text{NEXPTIME}^C$ where $C$ is the complexity class of solving $\exists:\text{LOC}_{[\text{dFA}]}$ (or $\exists:\text{ML}_{[\text{dFA}]}$). By Theorem 7.4, both of these problems are in $\text{EXPSPACE}$. Thus, the whole algorithm works in $2\text{-EXPSPACE}$. (Note that, $\text{EXPSPACE}$ is the best known upper bound even for $\exists:\text{LOC}_{[\text{dFA}]}$.) □

The following analysis makes use of a technique introduced in [31] for building the perfect automaton for $dFA$-designs.

**Definition 22.** Let $D = \langle \tau, B \rangle$ be an $R$-design where $B = B_0 T_1 B_1 \ldots T_n B_n$ is a kernel box. Together with $D^k$ we consider the string design $\hat{D}^k$ defined as follows. Let $\Sigma = \Sigma \uplus \{\sigma_0, \ldots, \sigma_n\}$ be an extension of $\Sigma$ and $\sigma(f_n) = \sigma_0 f_1 \sigma_1 \ldots f_n \sigma_n$ be the kernel string built by combining the new symbols with the functions of $B$. We denote by $\hat{D}^k = \langle \hat{\Omega}^k, \sigma \rangle$ the $k^{th}$ $dFA$-design built from $D$ where $\hat{\Omega}^k = \hat{\Omega}^k(\tau, u^k)$ is the perfect automaton built as described in [31]. □

The following lemma is a direct consequence of the definition of $\hat{\Omega}$ in [31].

**Lemma 7.5.** A typing $(\tau_n)$ is sound for $D^k$ iff it is sound for $\hat{D}^k$.

**Theorem 7.6.** Let $D = \langle \tau, B \rangle$ be a $dFA$-design and $(\tau_n)$ be a $D$-consistent typing. The following are equivalent:

1. $(\tau_n)$ is perfect for $D$;
2. $(\tau_n)$ is both local for $D$ and perfect for each $\hat{D}^k$.

Proof. (1) $\Rightarrow$ (2): If $(\tau_n)$ is perfect for $D$, then it is sound for each $D^k$, and by Lemma 7.5 sound for $\hat{D}^k$ as well. Suppose that $(\tau_n)$ is not local for some $\hat{D}^k$. There is a string $\sigma_0 u_1 \sigma_1 \ldots u_n \sigma_n \in [\hat{\Omega}^k]$ (all the strings have this form by definition) not captured by $\sigma(\tau_n)$. By Lemma 7.5, the string $w_0 u_1 w_1 \ldots u_n w_n$ belongs to $[\tau]$ and as $(\tau_n)$ is perfect, then each $u_i \in [\tau]$; contradiction. Suppose that $(\tau_n)$ is not perfect for some $\hat{D}^k$. There is a sound typing $(\tau'_n)$ for $\hat{D}^k$ not contained in $(\tau_n)$, but by Lemma 7.5, $(\tau'_n)$ is also sound for $D^k$, so $w(\tau'_n) \leq [\tau]$ again a contradiction because $(\tau_n)$ is perfect.

(2) $\Rightarrow$ (1): If $(\tau_n)$ is perfect for each $\hat{D}^k$ then, by Lemma 7.5, it is sound for each $D^k$. Suppose that it is not perfect for $D$, then there is a sound typing $(\tau'_n)$ not contained in $(\tau_n)$ such that, for some $k$, $w(\tau'_n) \leq [\tau]$ for the $k^{th}$ string $w_0, \ldots, w_n$. So $(\tau'_n)$ is sound for $D^k$ and also for $\hat{D}^k$. Contradiction. □

**Lemma 7.7.** $\exists:\text{PERF}_{[\text{dFA}]}^\beta$ is in $\text{coNP}$.
Proof. Let $D = \langle \tau, B \rangle$ be a dFA-design where $B = B_0 f_1 B_1 \ldots f_n B_n$ is a kernel box. We can decide in $\text{NP}$ whether $D$ does not admit any perfect typing by preforming the following steps:

1. **Guess**: four string-designs $D^{k_1}$, $D^{k_2}$, $D^{k_3}$, and $D^{k_4}$.
2. **Check**: answer “yes” ($D$ does not admit any perfect typing) if at least one of the following holds
   (a) $D^{k_1}$ does not admit any perfect typing;
   (b) $D^{k_2}$, $D^{k_3}$ have different perfect typings;
   (c) $D^{k_4}$ admits a perfect typing, say $(\tau_n)$, but it is not local for $D$.

Each of check (a), (b), and the first part of (c) require polynomial time \cite{31}. For the second part of check (c) we build $B(\tau_n)$ and prove that $B(\tau_n) < \tau$, namely $B(\tau_n) \cap \tau = \emptyset$. Notice that if the yes answer only depends on step (c) this means that $(\tau_n)$ is sound for each $D^k$, and so it is not possible that $B(\tau_n) > \tau$. Thus, as $\tau$ is a dFA, its complement has the same size and the intersection emptiness can be done in polynomial time as well.

**Corollary 7.8.** $\exists$-PERF $[\text{nFA-EDTD}]$ is in $\text{coNEXPTIME}$.

Proof. Let $D = \langle \tau, T \rangle$ be an nFA-EDTD-design. We build from $\tau$ its equivalent normalized version $\tau^d$ that, after all, is a dFA-EDTD of exponential size. By Corollary \cite{10} we polynomial-time reduce $\exists$-PERF $[\text{R-EDTD}]$ (for normalized R-EDTDs) to $\exists$-PERF $[\text{nFA}]$. So in our case we call $\exists$-PERF $[\text{dFA}]$. But, as $\tau^d$ may be exponentially larger, then, by adapting the upper bound of Lemma \cite{7.7} the whole algorithm works in $\text{coNEXPTIME}$.

**Theorem 7.9.** $\text{PERF}[\text{nFA-EDTD}]$ is in $\text{coNEXPTIME}$.

Proof. Let $D = \langle \tau, T \rangle$ be an nFA-EDTD-design, and $(\tau_n)$ be a $D$-consistent typing. Compute in $\text{coNEXPTIME}$ a perfect typing $(\tau'_n)$ if there is one. Transform $(\tau_n)$ into a dFA-EDTD-typing of exponential size. As $\text{EQUIV}[\text{dUTAs}]$ is in $\text{PTIME}$ then we can decide in $\text{EXPTIME}$ whether $(\tau_n)$ and $(\tau'_n)$ are equivalent.

Unfortunately, for ML-[nFA-EDTD] we do not have any good algorithm. Let $D = \langle \tau, T \rangle$ be an nFA-EDTD-design, $(\tau_n)$ be a maximal local typing for $D$, and $\kappa$ be de function induced by $(\tau_n)$ and $T$. At the moment, we do not even know whether there could be a (non-maximal) local typing $(\tau'_n) < (\tau_n)$ such that $\kappa' < \kappa$. If there is none, given a local typing $(\tau_n)$ and its induced function $\kappa$, then each maximal local typing that extends $(\tau_n)$ has to induce the same $\kappa$ as well. So we could compare the various $D^\kappa_x$ with $(\tau_n)$. But, the only know upper bound is given by the following theorem.

**Theorem 7.10.** ML-[nFA-EDTD] is in $2$-$\text{EXPSPACE}$.

Proof. Let $D = \langle \tau, T \rangle$ be an nFA-EDTD-design and $(\tau_n)$ be a $D$-consistent typing. We can check whether it is not maximal. Check in $\text{EXPTIME}$ whether it is local or not. So, build the normalized type $\tau^d$ from $\tau$. Guess a function $\kappa$ and check whether each $D^\kappa_x$ admits a local typing. This is in $2$-$\text{EXPSPACE}$ by Corollary \cite{7.4}
So, build the typing \((\tau'_n)\) induced by the box-designs. It may be an nFA-EDTD typing exponentially larger. Check whether \((\tau_n) < (\tau'_n)\). This can be done in 2-EXPTIME. So the algorithm works in 2-EXPSPACE and as this class is closed under complementation we also can decide ML\((nFA-EDTD)\) in it.

Finally, we consider the reduction from trees to boxes for \(∃\text{-ML}(dRE-EDTD)\). The difficulties affecting ML\((nFA-EDTD)\) (as we do not know whether there could be a local typing \((\tau'_n) < (\tau_n)\) such that \(κ' < κ\)) concern also the existential problem in case of dREs.

**Theorem 7.11.** \(∃\text{-ML}(dRE-EDTD)\) for normalized \(R\text{-EDTDs}\) is decidable by an oracle machine in PSPACE where \(C\) is the complexity class of solving the most difficult problem between \(∃\text{-ML}_{dRE} \exists\text{-LOC}_{dFA}\).

**Proof.** In this case we have to check two sources of maximality depending on the choice of \(κ\) and on the related box-designs. To do that, we guess a function \(κ\) (the candidate for a maximal local typing) and we check whether each induced box-design (i) admits a local typing, (ii) is maximal and (iii) is dRE-definable. So, we have to prove that each \(κ' > κ\) does not lead to any local typing. In particular:

1. Guess a functions \(κ\);
2. Prove that, for each node \(x\) of \(T\) with \(\text{lab}(x) ∈ Σ\), the answer of \(∃\text{-ML}_{dRE}\) over \(D^x_κ\) is “yes”;
3. Prove that, for each \(κ' > κ\), there is at least a node \(x\) of \(T\) with \(\text{lab}(x) ∈ Σ\) such that the answer of \(∃\text{-LOC}_{dFA}\) over \(D^x_{κ'}\) is “no”.

We just notice that there could be an exponential number of \(κ'\) to be enumerated and checked, as well as the number of calls to \(∃\text{-LOC}_{dFA}\). \(\square\)

**8. Conclusion**

As explained in the introduction, this work can serve as a basis for designing the distribution of a document. It would be interesting to extend our definitions and methods to richer types of web data. First, this would involve graph data and not just tree data. Then one should consider unordered collections and functional dependencies as in the relational model \([38, 39]\). Other dependencies and in particular inclusion dependencies would also clearly make sense in this setting \([40]\). More specific design methodology would also extend the techniques presented in this paper by considering concrete network configurations; this is left for future research.

Database design has a long history, see most database text book. Distributed database design has also been studied since the early days of databases, but much less, because distributed data management was limited by the difficulty to deploy distributed databases. The techniques that were developed, e.g., vertical and horizontal partitioning, are very different from the ones presented here because we focus on ordered trees and collections are not ordered in relational...
databases. We believe that traditional database studies even on mainly theoretical topics such as normal forms are also relevant in a Web setting. An interesting direction of research is to introduce some of these techniques in our setting.

In the paper, the focus was on local typing that forces verification to be purely local. More generally, it would be interesting to consider typings of the resources that would minimize the communications needed for type checking (and not completely avoid them). Moreover, it would be interesting to analyze cases where a kernel document may change from time to time by adhering to some global type which uses function symbols in the specification itself. We are investigating this direction. Let us give a short example exhibiting some of the new difficulties that would arise in case kernel document changes were taken into account. Consider the kernel string \( w = af \) and the type \( \tau = af?ba^+ \).

By directly applying the techniques proposed in this paper, it seems clear that \( f?ba^+ \) would be the perfect typing for this design. So, one extension of \( w \) may be \( afba \) (by attaching the tree \( fba \) complying with the perfect typing) which, in turn, represents a new kernel. But, this extension might still be extended, by attaching again tree \( fba \), to form \( afbaba \), since the first extension still contains a function call and the perfect typing defined for the remote resource should not vary. This last step could be performed several time. The language obtained by all possible extensions is defined by the type \( aF?(ba^+)^+ \), being clearly different from \( \tau \). The problem here is that \( \tau \) does not express directly a set of trees without taking into account a specific typing. New interesting questions might be: How to look for typings that are, in a sense, fixpoints w.r.t. the original type with functions? or How to avoid irregularities? or even Is the perfect typing still unique? Finally, interesting issues may also come from studying the impact of distributed typing (as studied here) on query optimization.

9. Acknowledgments

This work is a co-operation in the context of the ICT-2007.8.0 FET Open project Nr. 233599: Foundations of XML - safe processing of dynamic data over the Internet (FOX). Serge Abiteboul’s work was supported by the European Research Council Advanced Grant Webdam and by the French ANR Grant Docflow. Georg Gottlob’s work was supported by EPSRC grant EP/E010865/1 “Schema Mappings and Automated Services for Data Integration and Exchange”. Gottlob also gratefully acknowledges a Royal Society Wolfson Research Merit Award. Marco Manna acknowledges the support and hospitality of the Oxford-Man Institute of Quantitative Finance, where he worked as an Academic Visitor. Finally, the authors also want to thank Thomas Schwentick, Wim Martens, and Matthias Niewerth for useful discussions on the problem.

References


URL http://www.w3.org/TR/2006/REC-xml11-20060816/


URL http://repository.ust.hk/dspace/handle/1783.1/1783.1/738


URL http://relaxng.org/spec-20011203.html

URL http://www.grappa.univ-lille3.fr/tata


URL http://www.w3.org/TR/xml-fragment


