Expressiveness and complexity of XML Schema

Expressiveness and complexity of XML Schema

WIM MARTENS
and
FRANK NEVEN
Hasselt University and Transnational University of Limburg, School for Information Technology, Belgium
and
THOMAS SCHWENTICK
University of Dortmund, Department of Computer Science
and
GEERT JAN BEX
Hasselt University and Transnational University of Limburg, School for Information Technology, Belgium

---

The common abstraction of XML Schema by unranked regular tree languages is not entirely accurate. To shed some light on the actual expressive power of XML Schema, intuitive semantical characterizations of the Element Declarations Consistent (EDC) rule are provided. In particular, it is obtained that schemas satisfying EDC can only reason about regular properties of ancestors of nodes. Hence, w.r.t. expressive power, XML Schema is closer to DTDs than to tree automata. These theoretical results are complemented with an investigation of the XML Schema Definitions (XSDs) occurring in practice, revealing that the extra expressiveness of XSDs over DTDs is only used to a very limited extent. As this might be due to the complexity of the XML Schema specification and the difficulty to understand the effect of constraints on typing and validation of schemas, a simpler formalism equivalent to XSDs is proposed. It is based on contextual patterns rather than on recursive types and it might serve as a light-weight front end for XML Schema. Next, the effect of EDC on the way XML documents can be typed is discussed. It is argued that a cleaner, more robust, larger but equally feasible class is obtained by replacing EDC with the notion of 1-pass preorder typing (1PPT): schemas that allow to determine the type of an element of a streaming document when its opening tag is met. This notion can be defined in terms of grammars with restrained competition regular expressions and there is again an equivalent syntactical formalism based on contextual patterns. Finally, algorithms for recognition, simplification, and inclusion of schemas for the various classes are given.

Categories and Subject Descriptors: H.2.1 [**DATABASE MANAGEMENT**]: Logical Design; F.4.3 [**MATHEMATICAL LOGIC AND FORMAL LANGUAGES**]: Formal Languages

General Terms: Algorithms, Design, Languages, Standardization, Theory

Additional Key Words and Phrases: XML, XML Schema, validation

---

The present paper is the combined full version of [Martens et al. 2005] and [Bex et al. 2005].

1. INTRODUCTION

XML (eXtensible Markup Language) constitutes the basic format for data exchange on the Web [Bray et al. 2004]. Although the success of XML is largely due to its flexible nature, for many applications, it is important to constrain the structure of allowable documents by providing a schema. To date, the most widespread and commonly used schemas are DTDs. Their success is mostly of historic nature (DTDs are inherited from XML's predecessor SGML) and partly because of their simplicity. Unfortunately, DTDs are also limited in various ways [DuCharme 2002; Jelliffe 2001; Lee and Chu 2000]: they lack modularity, they have few basic types, and the referencing mechanism is quite restricted. Also, specification of unordered data is rather verbose and the expressiveness is severely limited. Many schema languages have been defined to address these shortcomings, to name just a few: XML Schema [Sperberg-McQueen and Thompson 2005], DSD [Klarlund et al. 2000], Relax NG [Clark and Murata 2001], Schematron [Jelliffe 2005]. Among these, XML Schema is the schema language supported by W3C and therefore receives the most attention. Although XML Schema directly addresses most of the shortcomings of DTDs, and in particular, is more expressive than DTDs, the exact expressiveness of XML Schema, and more importantly, whether the latter is adequate, remains unclear.

The main cause for the limited expressiveness of DTDs is that the content model of an element can not depend on the context of that element but only on the name of its tag. In formal language theoretic terms, DTDs define *local tree languages*. On an abstract level, XML Schema, just like Relax NG, obtains a higher expressive power by extending DTDs with a typing mechanism which allows to define types, possibly recursively, in terms of other types. In particular, and in contrast with DTDs, several types can be associated to the same element name. Whereas Relax NG corresponds to the robust and well-understood formalism of *unranked regular tree languages* [Brüggemann-Klein et al. 2001], XML Schema is less expressive as the XML Schema specification enforces an extra constraint: the *Element Declarations Consistent (EDC)* constraint. It essentially prohibits the occurrence of two different types with the same associated element name in the same content model.

In this paper, we investigate the impact of the EDC constraint on the expressiveness of XML Schema both from a theoretical and a practical perspective. We also suggest an alternative, theoretically and practically superior approach to achieve the goals that motivate the introduction of this constraint. Our characterizations show that, in terms of expressive power, XML Schema lies between DTDs and general tree automata. In fact, it turns out that, in contrast to what is generally assumed, in a sense, XML Schema is much closer to DTDs than to tree automata. This has both an effect on schema design and schema usage, as argued further on. We also investigate optimization problems for XML Schema, and show that they are easier to solve than the respective problems for tree automata.

We give an overview of the results of the paper in the order of their appearance. The presentation of the main results is separated from their proofs. This facilitates readers to skip the more technical sections (during their first pass). We start in Section 2 by introducing the necessary definitions concerning schemas and types, and discuss properties of DTDs in Section 3.

In Section 4, we investigate to what extent the features not present in DTDs are actually used in XML Schema Definitions (XSDs) occurring in practice: namespaces, import facilities, built in basic types, keys, and also the ability to use the same element name with different types. To this end, we harvested a corpus of XSDs from the web, including many high-quality schemas representing various XML standards. Concerning expressive power we were surprised that only 15% of the XSDs in our corpus use typing in a way that goes beyond the power of DTDs. Moreover, of this 15% the vast majority of the schemas use typing in its most simplistic form: types only depend on the parent context. Although it might indeed be the case that advanced expressiveness is not required in practice, another plausible explanation is that the actual modeling power of XSDs remains unclear to most users: the XML Schema specification is very hard to read and the effect of constraints on typing and validation is not fully understood. Thus, the average XML practitioner would benefit from a clear description of what kind of context dependencies can actually be expressed within XML Schema, and the implication of constraints such as EDC.

To address this issue, we propose two directions. We provide semantic and syntactical characterizations of the expressive power of schemas with the EDC constraint. This approach is pursued in Section 6 and 7. In particular, it is shown there that the EDC constraint is intimately connected to the ability to type trees in a top-down fashion. The characterizations provide different viewpoints on the expressiveness of XML Schema. One of them provides a tool that can be used to show that certain constructs are not definable by XSDs. The second direction is a simple pattern-based framework discussed in Section 5, much in the spirit of Schematron [Jelliffe 2005] and DSD [Klarlund et al. 2000]. The main difference is that a simple instantiation of this framework leads to a schema language with precisely the expressive power of (the core of) XSDs. The advantage of the pattern-based approach is that it makes explicit the way in which context-dependencies can be expressed rather than hiding it through the general use of recursive types restricted by the EDC constraint. The pattern-based approach can be migrated into a full fledged schema language in two ways: (1) as an extension of DTDs with contextual patterns; or, (2) as an extension of XML Schema itself (e.g., like SchemaPath [Coen et al. 2004], with the crucial difference that our extension is conservative).

Next, in Section 8, we turn to the question whether the EDC constraint is adequate for its purpose. For this, it is important to note that computing the semantics of a schema w.r.t. a document conceptually involves two tasks: (1) checking conformance w.r.t. the underlying grammar; and (2) assignment of types (also referred to as schema-validity assessment in [Thompson et al. 2004]). In the case of XML Schema, the two tasks are a bit entwined as types do not occur in the input document but have to be inferred by the schema validator. The EDC constraint is imposed to facilitate both tasks. Indeed, for a schema admitting EDC, there is a very simple one-pass top-down strategy to validate a document against that schema. Moreover, that strategy assigns a unique type to every element name. So, ambiguous typing (the possibility that there are several valid type assignments) is avoided. From a scientific viewpoint, however, it is not clear whether EDC is the most liberal constraint that allows for efficient validation and unique typing. One might argue

that the most liberal notion is to require that, when processing the document in a streaming fashion, the type of an element is assigned when its opening tag is met. We refer to the latter requirement as *1-pass preorder typing (1PPT)*. Although EDC ensures 1-pass preorder typing, it is not a necessary condition. More interestingly, it turns out that 1-pass preorder typing is a very robust notion with various clean semantical and syntactical characterizations. In particular, it can be defined in terms of XSDs with restrained competition regular expressions (introduced by Murata et al. [Murata et al. 2005]) and by an equivalent syntactical formalism based on contextual patterns.We therefore propose to replace the rather ad-hoc EDC and *unique particle attribution (UPA)* constraints by (a syntactical definition of) the 1-pass preorder typing requirement thereby obtaining the maximal expressiveness in terms of typing in a streaming fashion. In Section 8.7, we relate 1PPT and EDC with UPA. In particular, we argue that UPA and EDC do not imply each other. However, each of these constraints by itself already implies 1PPT (but they do not capture it). Hence, they are both not necessary to allow for fast typing. In Section 9, we give proofs for the equivalences stated in Section 8.

In Section 10, we turn to static analysis and optimization of schemas. In particular, we consider the complexity of and provide algorithms for the following problems:

(1) Recognition: Given an unrestricted XSD, check whether it admits EDC or 1PPT.
(2) Simplification: Given an unrestricted XSD, check whether it has an equivalent XSD of a restricted type and compute it, (restricted types being DTD or XSD admitting EDC or 1PPT).
(3) Containment: Given two XSDs D_1, D_2, does D_1 describe a sublanguage of D_2?

The above problems have direct practical applications to optimize schemas and to implement schema validators. Especially, our algorithm for Simplification could be used in schema translator software (like, for instance, Trang [Clark 2002]), to check whether a given schema can in fact be translated into an equivalent schema in another schema language. To date, Trang, for instance, translates any Relax NG schema directly into an equivalent unrestricted XSD even when the resulting XSD does not admit EDC. Sometimes, however, a more clever translation has to be used to get an equivalent XSD that admits EDC.

In Section 11, we discuss one-pass post-order typing: the type of an element is assigned when visiting its closing tag in a streaming fashion. We show that any unrestricted XSD can be rewritten into an equivalent one that admits one-pass post-order typing.

We conclude in Section 12. In particular, we present some detailed recommendations to improve the XML Schema specification.

Related Work. The analysis in the present work is in the same spirit as the one by Brüggemann-Klein and Wood who formalized the determinism constraint of SGML DTDs and provided a workable definition [Brüggemann-Klein and Wood 1998]. It is also closely related to the investigations of Murata et al. [Murata et al. 2005] who defined the concepts of single-type and restrained competition grammars

and provided corresponding validation algorithms but did not discuss semantical characterizations or optimization problems. The present paper is the combined full version of [Martens et al. 2005] and [Bex et al. 2005]. It is an attempt to seamlessly integrate the theoretical results of [Martens et al. 2005] with the discussion of their impact on the XML specification in [Bex et al. 2005]. As opposed to [Martens et al. 2005] we present full proofs for the results. On the other hand, we tried to lighten notation and reduce the number of theoretical concepts as much as possible to make the article more accessible.

2. SCHEMAS AND TYPES

In the present section, we provide the formal definitions of our abstractions of XML documents and schema languages.

2.1 Trees and tree languages

Since ordered trees serve as the logical data model for XML [Fernandez et al. 2005], we employ a tree based abstraction of XML documents. We focus in this work on the structure of XML documents and disregard data values, attributes, namespaces, and linking information. Figure 1 gives an example of (a) an XML document, (b) its tree representation with data values and (c) its tree representation without data values.

We disregard data values because we focus in this paper on the structural expressiveness of schema languages, i.e., the way in which schemas can restrict the shape or structure of XML documents. For the same reason we do not take special care of attributes and tacitly assume that all attributes are converted to non-nested elements. These restrictions are justified for our investigation as the Element Declaration Consistent constraint does not refer to attributes or data values.

More formally, we define the **associated tree t of an XML document** recursively as follows:

—The set of labels of t is the set of element names of the document.
—A document $\langle a\rangle w\langle/a\rangle$, where w contains only text has an associated tree with one node, labeled a.
—A document of the form $\langle a\rangle x_1 \cdots x_k\langle/a\rangle$, where the documents $x_1, \ldots, x_k$ have associated trees $t_1, \ldots, t_k$, has an associated tree with root a at which the trees $t_1, \ldots, t_k$ are attached, from left to right (see Figure 1 again).

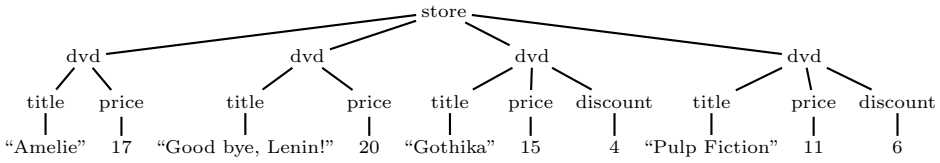
We denote the label of a node v in a tree t by $\mathrm{lab}^t(v)$. For a finite alphabet of element names Σ, we denote by $\mathcal{T}_\Sigma$ the set of all Σ-trees (trees with element names from Σ). Note that trees are unranked in the sense that every node can have an arbitrary number of children. For instance, in Figure 1, a store can sell an unlimited (but finite) number of DVDs. A **tree language** is a set of trees. For a gentle introduction into trees, tree languages and tree automata we refer to [Neven 2002a].

2.2 DTDs

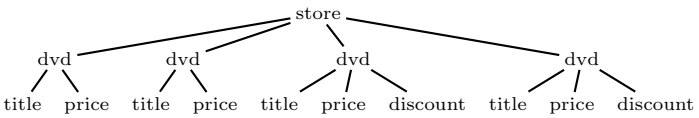
It is customary to abstract DTDs by *extended context-free grammars*, i.e., essentially by sets of rules of the form $a \to r$ where a is an element and r is a regular expression over the alphabet of elements. One element name is designated as the start symbol.

```
<store>
  <dvd>
    <title> "Amelie" </title>
    <price> 17 </price>
  </dvd>
  <dvd>
    <title> "Good bye, Lenin!" </title>
    <price> 20 </price>
  </dvd>
  <dvd>
    <title> "Gothika" </title>
    <price> 15 </price>
    <discount> 4 </discount>
  </dvd>
  <dvd>
    <title> "Pulp Fiction" </title>
    <price> 11 </price>
    <discount> 6 </discount>
  </dvd>
</store>
```

(a) Example document



(b) Tree with data values



(c) Tree without data values

Fig. 1. An example of an XML document and its tree representations.

```
<!ELEMENT store        (dvd+)>
<!ELEMENT dvd          (title, price, discount?)>
<!ELEMENT title        (#PCDATA)>
<!ELEMENT price        (#PCDATA)>
<!ELEMENT discount     (#PCDATA)>
```

Fig. 2. A DTD for which the document in Figure 1 is valid.

Example 2.1. The DTD of Figure 2 is represented by the following rules, where `store` is the start symbol:

$$\begin{aligned} \texttt{store} &\rightarrow \texttt{dvd}\,\texttt{dvd}^* \\ \texttt{dvd} &\rightarrow \texttt{title}\,\texttt{price}\,(\texttt{discount} + \varepsilon) \end{aligned}$$

□

Clearly, a DTD defines a set of allowed trees, hence a tree language. The DTD in Figure 2, for instance, defines the set of trees where the root is labeled with `store`; the children of `store` are `dvd` elements; every `dvd` element has a `title`, `price`, and an optional `discount` child.

The notion of a DTD is formalized as follows.

Definition 2.2. A **DTD** is a triple (Σ, d, s_d) where Σ is a finite alphabet (*the*

element names), d is a function that maps Σ-symbols to regular expressions and $s_d \in \Sigma$ is the *start symbol*. We usually abbreviate (Σ, d, s_d) by d when Σ and s_d are clear from the context. A (finite) tree t **is valid w.r.t.** d (or **satisfies** d) if its root is labeled by s_d and, for every node with label a, the sequence $a_1 \cdots a_n$ of labels of its children is in the language defined by $d(a)$. By $L(d)$ we denote the set of trees that satisfy d.

The regular expression associated to an element name is sometimes also called its *content model*.

2.3 DTDs plus types

As discussed in the introduction, the expressive power of DTDs can be extended by adding types, as, e.g., in XML Schema [Thompson et al. 2004] and Relax NG [Clark and Murata 2001]. Types are always from a finite set and each type is associated with a unique element name. The designated start symbol has only one possible type. The notion of *extended DTDs (EDTDs)* was introduced[1] by Papakonstantinou and Vianu [Papakonstantinou and Vianu 2000].

Definition 2.3 *[Papakonstantinou and Vianu 2000; 2003].* An **extended DTD** (EDTD) is a tuple $D = (\Sigma, \Delta, d, s_d, \mu)$, where Δ is a finite set of *types*, (Δ, d, s_d) is a DTD and μ is a mapping from Δ to Σ. A tree t is **valid w.r.t.** D (or **satisfies** D) if $t = \mu(t')$ for some $t' \in L(d)$ (where μ is extended to trees). We call t' a **witness** for t.

Intuitively, a tree satisfies an EDTD if there exists an assignment of types to all nodes such that the typed tree is a derivation tree of the underlying grammar. The following example displays an EDTD for the tree in Figure 1(c).

Example 2.4. Consider the following EDTD:

$$\begin{aligned} &\texttt{store} &&\rightarrow (\texttt{reg-dvd} + \texttt{dis-dvd})^*\texttt{dis-dvd}(\texttt{reg-dvd} + \texttt{dis-dvd})^* \\ &\texttt{reg-dvd} &&\rightarrow \texttt{title price} \\ &\texttt{dis-dvd} &&\rightarrow \texttt{title price discount} \end{aligned}$$

Here, `reg-dvd` and `dis-dvd` are types that are associated to `dvd` elements, while all other types are associated with the element of the same name: e.g., the type `store` corresponds to a `store` element.

Intuitively, `reg-dvd` defines ordinary DVDs while `dis-dvd` defines DVDs on sale. The first rule specifies that there has to be at least one DVD on discount. The tree in Figure 1(c) satisfies this EDTD as assigning `reg-dvd` and `dis-dvd` to the left and right `dvd`-node, respectively, gives a derivation tree of the grammar. □

When the validity of a tree t is witnessed by a tree t' then we call the label of a node v in t' its **type** with respect to this validation. The set of trees defined by D is denoted $L(D)$. For notational simplicity, we assume in proofs and formal statements always that types are of the form a^i with $a \in \Sigma$, $i \in \mathbb{N}$ and $\mu(a^i) = a$.

[1] Papakonstantinou and Vianu used the term *specialized DTD* as types *specialize* tags. We prefer the term *extended DTD* as it expresses more clearly that the power of the schemas is amplified.

Note that EDTDs have a single root type; only labels below the root can have multiple types. The EDTD of Example 2.4 has the types `store`, `reg-dvd`, `dis-dvd`, `title`, `price` and `discount`. Further, $\mu(\texttt{reg-dvd}) = \mu(\texttt{dis-dvd}) = \texttt{dvd}$ and μ is the identity, otherwise.

```
<xs:element name="store">
  <xs:complexType>
    <xs:sequence>
      <xs:choice minOccurs="0" maxOccurs="unbounded">
        <xs:element name="dvd" type="reg-dvd"/>
        <xs:element name="dvd" type="dis-dvd"/>
      </xs:choice>
      <xs:element name="dvd" type="dis-dvd"/>
      <xs:choice minOccurs="0" maxOccurs="unbounded">
        <xs:element name="dvd" type="reg-dvd"/>
        <xs:element name="dvd" type="dis-dvd"/>
      </xs:choice>
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

Fig. 3. A fragment of an XSD (violating EDC) corresponding to the EDTD of Example 2.4.

In Figure 3, a fragment of an XSD corresponding to the rule for `store` is depicted. We note that the XSD is not syntactically correct because it violates the Element Declarations Consistent constraint. Roughly, the constraint forbids the occurrence of different types associated to the same element name in the same content model. So, the occurrence of both `reg-dvd` and `dis-dvd` associated to the same element name `dvd` is a clear violation of this constraint. A formalization and a detailed discussion of this constraint is provided in Section 6.

We call a tree language **homogeneous** if all its trees have the same root label. It should be clear that EDTDs can only express homogeneous tree languages. From a structural perspective, EDTDs express exactly the *homogeneous regular tree languages*, a similarly robust class as the regular string languages [Brüggemann-Klein et al. 2001]. In particular, EDTDs are as expressive as unranked tree automata. For definitions of such automata we refer the reader to Section 10 and, e.g., [Brüggemann-Klein et al. 2001; Neven 2002b]. It should be noted that the formal underpinnings of the schema language Relax NG are also based upon regular tree languages. As we will only talk about homogeneous tree languages we will mostly drop the term homogeneous. For the purpose of the paper, whenever we say *(homogeneous) regular tree language* T in the sequel, it can be interpreted as *there is an EDTD* D *such that* $L(D) = T$.

3. PROPERTIES OF DTDS

In this section, we reconsider some simple properties of DTDs. In particular, we discuss validation and a closure property. The latter property provides a tool to prove that certain tree languages are not definable by DTDs, and, hence, gives insight into the expressiveness of the latter. In Section 6 and Section 8, we discuss similar closure properties for more restrictions of EDTDs.

3.1 Validation of DTDs

Validation of a document against a DTD d simply boils down to testing local consistency: does the string formed by the labels of the children of every a-labeled element satisfy the associated regular expression $d(a)$? No notion of typing is available. To ensure efficient validation, regular expressions in right-hand sides of rules are required to be *deterministic* [Bray et al. 2004], Appendix E (also referred to as *one-unambiguous* [Brüggemann-Klein and Wood 1998]). Intuitively, a regular expression is deterministic if, when processing the input from left to right, it is always determined which symbol in the expression matches the next input symbol. We discuss the latter notion a bit more formally as it returns in the specification of XML Schema in the form of the *Unique Particle Attribution (UPA)* rule. For a regular expression r over elements, we denote by $\overline{r}$ the regular expression obtained from r by replacing, for each i, the ith a-element in r (counting from left to right) by a_i. For example, when $r = (a+b)^*ac(b+c)^*$, then $\overline{r}$ is $(a_1+b_1)^*a_2c_1(b_2+c_2)^*$.

Definition 3.1. A regular expression r is **one-unambiguous** iff there are no strings wa_iv and wa_jv' in $L(\overline{r})$ so that $i \neq j$.

Example 3.2. The regular expression $ab + aa$ is not one-unambiguous. Indeed, $L(a_1b_1 + a_2a_3)$ contains the strings a_1b_1 and a_2a_3, and $1 \neq 2$. Here, w is the empty string. The expression $a(b + a)$ on the other hand, is one-unambiguous. Indeed, $L(a_1(b_1 + a_2))$ only contains the strings a_1b_1 and a_1a_2, and it is easy to verify that the condition is not violated. Note that $a(b + a)$ denotes the same language as $ab + aa$. □

In contrast to what the previous example might suggest, Brüggemann-Klein and Wood showed that not every regular expression can be rewritten into an equivalent one-unambiguous one [Brüggemann-Klein and Wood 1998]. So the allowed class of regular expressions is a strict subset of the class of all regular languages.

3.2 Subtree exchange

Papakonstantinou and Vianu [Papakonstantinou and Vianu 2000] provided a characterization of the structural expressive power of DTDs. They considered a more relaxed notion of DTDs without the requirement of one-unambiguous regular expressions. They show that a regular tree language T of trees is definable by such a DTD if and only if T has the following closure property: if two trees t_1 and t_2 are in T, and there are two nodes v_1 in t_1 and v_2 in t_2 with the same label, then the trees obtained by exchanging the subtrees rooted at v_1 and v_2 are also in the set T. We refer to this property as **label-guarded subtree exchange** and we illustrate it in Figure 4.

Because of this characterization, the classes of XML documents defined by DTDs are also referred to as **local classes** (cf. [Murata et al. 2001]): the content of a node only depends on the label of that node and hence, the dependency is local. The characterization can be used to prove that certain languages can *not* be expressed by a DTD as explained in the following example.

Example 3.3. Suppose that we want to put the extra constraint on the DTD of Figure 2 requiring the presence of at least one DVD on discount. Then we get a

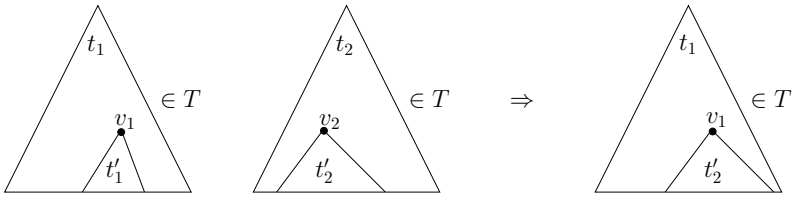
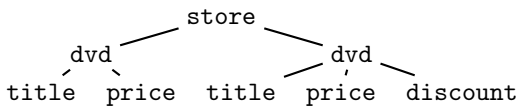


Fig. 4. Label-guarded subtree exchange. Nodes v_1 and v_2 are both labeled with the same label.

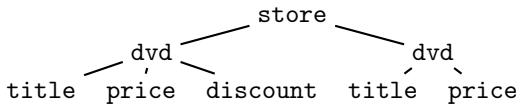
language that is *not* definable by a DTD anymore. We can prove this by applying the above characterization. Indeed, the trees

$t_1 :=$

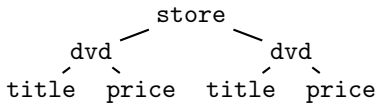


and

$t_2 :=$



are in the language, but the tree



which is obtained from t_1 by replacing its second subtree by the second subtree of t_2, is *not* in the language. □

4. A PRACTICAL STUDY OF XSDS

A variety of sources [DuCharme 2002; Jelliffe 2001; Lee and Chu 2000] discuss the many drawbacks of DTDs: no modularity, no XML syntax, limited basic types, restricted referencing mechanism, verbose specification of unordered data, and limited expressiveness (definition of an element cannot depend on its context). Most of these concerns have been addressed by the XML Schema specification: namespaces and import facilities have been added; an extensive number of built in basic types as well as means to fine tune them by restriction are provided; XML Schema supports key and referential integrity; the `all` construct allows to specify unordered content; and finally, different types can be allowed for the same element name. Of course, this raises the question to what extent the added features are actually used in practice. To this end, we studied a corpus of 819 XSDs harvested from the Web. Among the XSDs gathered, 93 were retrieved via the Cover Pages [Cover 2005].[2] Hence, a substantial number of high-quality XSDs representing various standards

[2]A previous study only focused on the Cover Pages and also investigated the structure of used regular expressions [Bex et al. 2004].

such as the XML Schema Specification, XHTML, UDDI, RDF and others are represented in the corpus. Unfortunately this number is rather small, so the corpus was enlarged to its present size of 819 XSDs using Google's web services to retrieve an additional 726 XSDs.[3]

The results concerning the use of syntactical features are summarized in Table I. From this table one can conclude that XML Schema's simpleType library and the ability to place restrictions on those are heavily used. Derivation in the sense of the object orientation paradigm is only used in about 1/5 of all XSDs. Modularity by way of imports and (non-trivial) namespaces is fairly important as well. Uniqueness and key references are quite uncommon.

feature	**% of XSDs**
derivation	
simpleType extension	18.9
simpleType restriction	45.5
complexType extension	20.7
complexType restriction	3.6
abstract attribute	9.8
final attribute	0.9
block attribute	0.0
fixed attribute	6.4
substitutionGroup	6.4
redefine	1.0
interleaving	
`xs:all`	5.5
modularity	
namespaces	12.1
import	27.7
linking	
key/keyref	4.1
unique	2.9

Table I. XML Schema features use in the corpus

As explained above, XSDs employ types to increase expressiveness beyond DTDs. The question remains whether XSDs occurring in practice actually use this feature. That is, what percentage of found XSDs are not structurally equivalent to a DTD? Unfortunately, out of the corpus of 819 harvested XSDs only 225 remain on which IBM's SQC [Fokoué and Schloss 2004] reports no errors.[4] Although syntactical correctness is less critical in testing for presence or absence of syntactical features, it is mandatory for the expressiveness analysis which is more semantical in nature. It is impossible to automatically guess for every syntactically incorrect XSD what its designer intended.

[3]The study was performed in September 2004.

[4]Even worse, already 70% of the XSDs from the Cover pages do not pass the syntax checker SQC. In this respect it is interesting to note that Sahuguet reported similar findings on the sheer abundance of syntactically incorrect DTDs [Sahuguet 2000].

It turns out that out of the remaining 225 XSDs, 192 (85%) are in fact structurally equivalent to a DTD: at most one type is associated to every element name.[5] So only 33 XSDs (15%) use the typing mechanism to actually define non-local classes of XML documents. Surprisingly, in 30 XSDs, types only depend on the parent context.

Example 4.1. We give an example of an EDTD where types only depend on the parent context:

$$\begin{array}{ll} \texttt{store} & \rightarrow \texttt{regulars discounts} \\ \texttt{regulars} & \rightarrow (\texttt{reg-dvd})^* \\ \texttt{discounts} & \rightarrow \texttt{dis-dvd}\ (\texttt{dis-dvd})^* \\ \texttt{reg-dvd} & \rightarrow \texttt{title price} \\ \texttt{dis-dvd} & \rightarrow \texttt{title price discount} \end{array}$$

where $\mu(\texttt{reg-dvd}) = \mu(\texttt{dis-dvd}) = \texttt{dvd}$. Here, the content model of a `dvd` is `title price` when it occurs as the child of a `regulars` element, and it is `title price discount` when it occurs as the child of a `discounts` element. It should be noted though that the tree language described by this EDTD is different from the one of Example 2.4 as it uses additional tags, `regulars` and `discounts`. □

In contrast, in Example 2.4 types depend on (the subtrees) at all siblings.

So, although non-trivial typing is moderately used in practice, it is almost exclusively used in its most simplistic form: dependence on the label of the parent. Recall that for DTDs, the type of an element is its name. Basically, there are two possible explanations for the above observation. Either, advanced modeling power as allowed by EDTDs is not necessary in practice. Or, users are simply not aware of what kind of schemas can be expressed by XSDs. In Section 6, we address the latter concern as we provide several characterizations which give insight into what is theoretically possible when using XSDs. The former possibility is analyzed below in more detail.

In the remaining 3 XSDs, types depend on the grand- or the great grand-parent context. We discuss an abstraction of one of them as an EDTD:

Example 4.2.

$$\begin{array}{rcl@{\qquad}rcl} a & \rightarrow & b + c & h^1 & \rightarrow & j^1 \\ b & \rightarrow & e\ d^1\ f & h^2 & \rightarrow & j^2 \\ c & \rightarrow & e\ d^2\ f & j^1 & \rightarrow & k\ \ell \\ d^1 & \rightarrow & g\ h^1\ i & j^2 & \rightarrow & m\ n \\ d^2 & \rightarrow & g\ h^2\ i & & & \end{array}$$

The interpretation of the example above is simple: a j^1 element can only occur as the great grandchild of a b element while a j^2 element can only occur as the great grandchild of a c element. □

Two extreme approaches can be used to code the abstract example above in an XSD. On the one hand, one can use the "Russian doll" model, i.e. using anonymous type

[5] Actually, we encountered one XSD using types to define a local language. The corresponding EDTD is of the form: $a^1 \rightarrow b$, $a^2 \rightarrow b$ where the types differ semantically.

definitions within type definitions. In an abstract syntax the latter reduces to the rules

$$b \to ed[gh[j[k\ell]]i]f \quad \text{and} \quad c \to ed[gh[j[mn]]i]f,$$

where the type definition of the element b encapsulates that of d^1 which in turn defines that of h^1 that finally contains j^1's definition. The alternative is to flatten the XSD as has been done in Example 4.2, but this leads to "artificial types" such as d^1 and h^1 that only exist to pass down the information that their parent and grandparent was a b-element. It is obvious that in practice both approaches are mixed to a certain extent. However, both lead to duplication of definitions, making maintenance and further development of an XSD much harder. In the next section, we present a pattern-based alternative to XSDs allowing to make context dependencies explicit.

5. PATTERN-BASED SCHEMA LANGUAGES

We address the concerns mentioned in the previous section in two ways: (1) we give in Section 6 several equivalent characterizations explaining the meaning of the EDC constraint; and, (2) we propose in the present section a general framework for XML schema languages, *pattern-based schemas*, that allows types to depend on the labels of ancestors.

The proposed framework is related to the paradigm of *contextual patterns* upon which languages like Schematron [Jelliffe 2005] and DSD [Klarlund et al. 2000] are based. These pattern-based schemas no longer require to define types in terms of types, which seems to be perceived as a challenge by the average XSD author, but still allow to access the full power offered by XML Schema. Duplication of definitions would be reduced as well since dependencies on ancestor labels can straightforwardly be declared rather than being passed down via types. Our framework can be instantiated in various ways, resulting, e.g., in all tree languages definable by an XSD or all tree languages which can be efficiently typed in a streaming fashion as detailed in Sections 6 and 8, respectively.

5.1 Pattern-based schemas by example

To enlarge flexibility, in a first stage, we only use patterns in an abstract way. Therefore, we assume a pattern language where each pattern associates with every tree a set of selected nodes. We will consider linear XPath expressions, i.e., using the axes `/` (child) and `//` (descendant) only, regular expressions, and full XPath. Before we define pattern-based schemas more formally, we take a look at an example.

Example 5.1. The following pattern-based schema uses linear XPath as a pattern language and describes the EDTD of Example 4.1. It describes a store, in which regular and discount DVDs are present. A discount DVD has a tag `discount`, whereas a regular DVD does not. This distinction is described by referring to the path from the root to a DVD-element:

$$
\begin{array}{rcl}
\texttt{//store} & \rightarrow & \texttt{regulars discounts} \\
\texttt{//regulars} & \rightarrow & \texttt{dvd}^* \\
\texttt{//discounts} & \rightarrow & \texttt{dvd dvd}^* \\
\texttt{//regulars/dvd} & \rightarrow & \texttt{title price} \\
\texttt{//discounts/dvd} & \rightarrow & \texttt{title price discount}
\end{array}
$$

This schema declares four elements: `store`, `regulars`, `discounts`, and `dvd`. For `dvd` there are two rules. The first one defines regular dvds while the second one defines discount dvds. As an example, the expression of the last line holds at a node v, if (1) v is labeled with `dvd` and its parent is labeled with `discounts` and (2) its children are labeled by `title`, `price`, `discount`, from left to right.

□

5.2 A formal definition of pattern-based schemas

We next give a more formal definition of pattern-based schemas. To this end, let $\mathcal{P}$ be a pattern language defining unary patterns. That is, each pattern $\varphi \in \mathcal{P}$ associates with every tree t a set of selected nodes, which we denote by $\varphi(t)$.

Definition 5.2. A **$\mathcal{P}$-schema** is a pair $S = (\Sigma, R)$ where R is a finite set of rules of the form $\varphi \rightarrow s$. Here, $\varphi \in \mathcal{P}$ is a pattern, and s is a regular expression over Σ. A tree t is **valid** w.r.t. S if the label of every node belongs to Σ and for every node v of t there is a rule $\varphi \rightarrow s$ such that $v \in \varphi(t)$ and the children of v match the regular expression s.

Remark 5.3. It should be noted here that there are some possible variants in the definition of pattern-based schemas and their semantics. For instance, as just defined, the semantics has an *existential* nature. Each node has to match at least one rule. A *universal* semantics could require that for for each rule $\varphi \rightarrow s$, for each node v in $\varphi(t)$ the children of v match s. For pattern languages closed under the Boolean operations, the two semantics are equally expressive.

Example 5.4. An XPath-schema equivalent to the EDTD of Example 4.2 is the following:

$$
\begin{array}{lll}
a \rightarrow b + c & & h \rightarrow j \\
b \rightarrow e\,d\,f & & //b//j \Rightarrow k\,\ell \\
c \rightarrow e\,d\,f & & //c//j \Rightarrow m\,n \\
d \rightarrow g\,h\,i & &
\end{array}
$$

Note that for brevity, we write $a \rightarrow b + c$ rather than the more correct $//a \rightarrow b + c$.
□

We note that with an expressive pattern language, such as full XPath, the expressive power of P-schemas may extend that of XML schema. For example, in [Fiorello et al. 2004] an approach called DTD++ 2.0 is introduced that allows to define schemas which are then translated into SchemaPath [Coen et al. 2004] which is strictly more expressive than XML Schema and requires a transformation of the XML documents prior to validation. In this paper, we present in Section 6 an instantiation of P-schemas whose expressive power is at most that of XML schema. Our approach therefore avoids the overhead of translating XML documents and leverages the use of existing XML Schema implementations and tools.

5.3 Pattern-based schemas in practice

The usefulness of an XML schema language requires more than an abstract framework. Therefore, we discuss next how we can migrate pattern-based schemas into a full fledged schema language. Rather than proposing yet another schema language we stipulate how existing languages and proposals can be adapted.

Several approaches guided by our practical study are conceivable. We suggest a two-pronged approach:

—on the one hand an extension to the DTD specification for those most comfortable with this formalism which probably includes inexperienced or new users,

—on the other hand, an extension of XML Schema more suited for advanced users.

However, these seemingly different approaches converge behind the scenes since schemas developed both according to the DTD extension and to the XML Schema extension can be translated into an XSD which is valid with respect to the current XML Schema specification. We next discuss both proposals in more detail.

5.3.1 *Enhancing DTDs.* The most direct approach is to enhance DTDs to the formalism of pattern-based schemas as exemplified in Examples 5.5 and 5.6. To increase readability we allow to specify names to patterns.

Pattern declarations could be of the form

```
<!PATTERN name pattern-expression (regular-expression)>.
```

Example 5.5. The schema of Example 5.1 could be represented as follows.

```
<!ELEMENT store (regulars, discounts)>
<!ELEMENT regulars (dvd*)>
<!ELEMENT discounts (dvd dvd*)>
<!PATTERN regular-dvd "//regulars/dvd" (title price)>
<!PATTERN discount-dvd "//discounts/dvd" (title price discount)>
```

□

Example 5.6. The real world XSD of Example 4.2 can be rewritten as the following enhanced DTD.

```
<!ELEMENT a (b | c)>
<!ELEMENT b (e, d, f)>
<!ELEMENT c (e, d, f)>
<!ELEMENT d (g, h, i)>
<!ELEMENT h (j)>
<!PATTERN j1 "//b//j" (k, l)>
<!PATTERN j2 "//c//j" (m, n)>
```

□

It is clear that the representation in the examples is much more compact than the corresponding XSDs, and that duplicate definitions have been avoided altogether. Note that the examples use "linear XPath" expressions, incorporating only the axes child and descendant. The results mentioned in Section 4 suggest that this expressiveness is sufficient to structurally capture the XSDs occurring in practice, though the power of full regular expressions is needed to capture all of XML Schema

(cf. Section 6.4). Thus, one can limit oneself to the abbreviated syntax ('/' and '//') which substantially contributes to the transparency of the expressions.

To make enhanced DTDs practically useful, often used features like simple types and namespaces should be added as well (cf. Table I). Proposals for such additions already exist [Vitali et al. 2003; Buck et al. 2000] and can easily be incorporated. Both focus heavily on the addition of data types to DTDs. The former (DTD++ 1.0) also introduces namespaces and complex objects. To the best of our knowledge we are the first to justify enhancements to DTDs both by a practical study (Section 4) and a theoretical analysis (Section 7). Indeed, in strong contrast to DTD++ 1.0, the restriction to pattern-based schemas can structurally define *all* XSDs (Theorem 7.1) and can therefore act as a complete front-end for XML Schema.

5.3.2 *A conservative extension of XML Schema.* The second option is to extend the XML Schema specification in such a way that element type definitions are context dependent. A syntactic approach using conditional alternatives similar to SchemaPath [Coen et al. 2004] is suggested. However, rather than full XPath expressions, conditions would be limited to linear XPath (or general regular expressions) so that the expressive power of XML Schema is not exceeded. Whereas the enhanced DTDs are more expressive than traditional DTDs, extended XML Schemas provide only syntactic sugar to ease the development and make XML Schema more legible and easier to maintain since a lot of definition duplications can be eliminated.

Example 5.7. The essential fragment of Example 5.1 can be rewritten as an extended XSD as follows:

```
<xs:element name="j">
  <xs:alt cond="//regulars/dvd" type="regular-dvd"/>
  <xs:alt cond="//discounts/dvd" type="discount-dvd"/>
</xs:element>
```

□

Example 5.7 shows a *conditional element definition*: element `dvd` is of type `regular-dvd` (`discount-dvd`) if it has a `regulars` (`discounts`) parent.

6. EDC AND ANCESTOR-BASED SCHEMAS

As already mentioned in the introduction, XML Schema does not capture all tree languages that can be described by extended DTDs (i.e., the regular tree languages). In particular, the EDC (Element Declarations Consistent) and the UPA (Unique Particle Attribution) constraint must be fulfilled. In this section, we formalize EDC in the form of a certain kind of EDTDs and give several equivalent characterizations of the resulting class of tree languages. One of these characterizations is in terms of pattern-based schemas. Together, these characterizations provide a clear view of the effect of the EDC constraint on the expressiveness of XSDs and typing algorithms.

To enhance readability of this section, we postpone most of the equivalence proofs to Section 7.

6.1 A formalization of EDC: single-type EDTDs

Murata et al. [Murata et al. 2001] presented a formalization of the EDC rule, which we state here in terms of EDTDs.[6] Roughly, the constraint forbids the occurrence in the same definition of elements with the same name but different types. For instance, the XSD of Figure 3 is not allowed as the two types `reg-dvd` and `dis-dvd` occur in the same rule.

Definition 6.1. An EDTD $(\Sigma, \Delta, d, s_d, \mu)$ is **single-type** if in no regular expression two types $\tau \neq \tau'$ with $\mu(\tau) = \mu(\tau')$ occur.

In the remainder of the paper, we use the terms *EDC* and *single-type* interchangeably. The EDTD of Example 2.4 is not single-type as both `reg-dvd` and `dis-dvd` occur in the rule for `store`. Example 4.1 shows a single-type EDTD. Although there are two types `reg-dvd` and `dis-dvd` for `dvd`, they occur in different rules.

6.2 Ancestor-based types

As remarked by Murata et al. [Murata et al. 2001], the definition of single-type EDTDs induces a very simple top-down typing algorithm which assigns a unique type to every node: First, the unique type is assigned to the root. Next, for each interior node u with type a^i, the algorithm finds the corresponding rule $a^i \to r$ and checks whether the children of u match $\mu(r)$, i.e., the regular expression obtained from r by replacing every type by its corresponding element name. If this fails the tree is rejected. Otherwise, as the EDTD is single-type, to each child a unique type can be assigned. The tree is then accepted if this process terminates at the leaves without any rejection.

This algorithm immediately implies that the type of a node only depends on its ancestors. We formalize this as follows. By anc-str$^t(u)$ we denote the sequence of labels on the path from the root to u including both the root and u itself (cf. Figure 6).

Definition 6.2. We say that an EDTD $D = (\Sigma, \Delta, d, s_d, \mu)$ **has ancestor-based types** if there is a function $f : \Sigma^* \to \Delta$ such that, for each tree $t \in L(D)$,

—t has exactly one witness t', and

—t' results from t by assigning to each node v the type $f(\text{anc-str}^t(v))$.

It is now easy to prove the following result:

PROPOSITION 6.3. *When a tree language T is definable by a single-type EDTD, then it has ancestor based types.*

PROOF. Let T be defined by the single-type EDTD $D = (\Sigma, \Delta, d, s_d, \mu)$. Then define f inductively as follows: $f(\mu(s_d)) = s_d$. Further, for any string $w \cdot a \cdot b$ with $w \in \Sigma^*$ and $a, b \in \Sigma$, $f(w \cdot a \cdot b) = b^j$ where b^j occurs in $d(a^i)$ and $f(w \cdot a) = a^i$. As $d(a^i)$ is single-type, f is well-defined and induces a unique typing. Thus, the requirements of Definition 6.2 are satisfied. □

We note that there are EDTDs which are not single-type but have ancestor-based types. But we will see in Section 7, that such EDTDs always have an equivalent single-type EDTD.

[6] [Murata et al. 2001] used the equivalent model of tree grammars instead of EDTDs.

6.3 A characterization of EDC by a subtree-exchange property

We introduce a tool to show that certain schemas are not definable by EDTDs admitting EDC. We recall the notion of label-guarded subtree exchange from Subsection 3.2 which characterized the class of DTD-definable languages. A similar characterization holds for single-type EDTDs.

To this end, we denote by $t_1[u \leftarrow t_2]$ the tree obtained from a tree t_1 by replacing the subtree rooted at node u of t_1 by t_2. By $\text{subtree}^t(u)$ we denote the subtree of t rooted at u.

Definition 6.4. A tree language T is **closed under ancestor-guarded subtree exchange** if the following holds. Whenever for two trees $t_1, t_2 \in T$ with nodes u_1 and u_2, respectively, $\text{anc-str}^{t_1}(u_1) = \text{anc-str}^{t_2}(u_2)$ then $t_1[u_1 \leftarrow \text{subtree}^{t_2}(u_2)] \in T$.

This definition is illustrated in Figure 5.

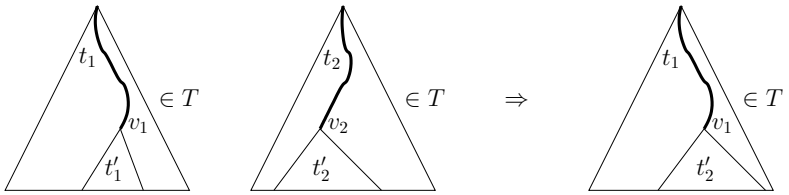


Fig. 5. Ancestor-guarded subtree exchange.

It is easy to see that ancestor-based types imply closure under ancestor-guarded subtree exchange:

PROPOSITION 6.5. *When an EDTD D has ancestor-based types then $L(D)$ is closed under ancestor-guarded subtree exchange.*

PROOF. Let T be defined by an EDTD $D = (\Sigma, \Delta, d, s_d, \mu)$ with ancestor-based types. Let t_1, t_2 be in T and let u_1 and u_2 be nodes in t_1 and t_2, respectively, with $\text{anc-str}^{t_1}(u_1) = \text{anc-str}^{t_2}(u_2)$. Let t'_1 and t'_2 be the unique witnesses for t_1 and t_2, respectively. As the label of u_1 in t'_1 and the label of u_2 in t'_2 are determined by $\text{anc-str}^{t_1}(u_1) = \text{anc-str}^{t_2}(u_2)$, they are the same. Hence, by replacing the subtree rooted at u_1 in t'_1 with the subtree rooted at u_2 in t'_2 we get a tree $t' \in L(d)$. Therefore, $\mu(t') = t_1[u_1 \leftarrow \text{subtree}^{t_2}(u_2)]$ is in T, as required. □

Again, the converse does not hold literally, but if an EDTD defines a tree language closed under ancestor-guarded subtree exchange, it always has an equivalent single-type EDTD. The proof is non-trivial and appears in Section 7.

As an immediate consequence of Proposition 6.5, the language we considered in Example 3.3 is not definable by a single-type EDTD. Note that the counterexample can be constructed in exactly the same manner. On the other hand, the language defined by the single-type EDTD in Example 4.1 is not definable by a DTD, so single-type EDTDs are strictly more expressive than DTDs. As a matter of fact, it can be decided whether a given EDTD is equivalent to a single-type EDTD. For instance, the non single-type EDTD $a \to b^1 b^2$, $b^1 \to c$, $b^2 \to c$ is clearly equivalent

to the DTD (and, hence, single-type EDTD) consisting of the rules $a \to bb$ and $b \to c$. The complexity of this problem is considered in Section 10.

The importance of the characterization of single-type EDTDs by a subtree-exchange property stems from the fact that inexpressibility results can be formally proved rather than vaguely stated. For instance, a shortcoming recently attributed to XSDs is their inability to express certain co-constraints [Coen et al. 2004]. An example of such a co-constraint is the following: a `store`-element can only have a `dvd`-element with `discount` child if it also has a `dvd`-element without a `discount` child. Using the ancestor-guarded subtree exchange property, it is very easy to prove that this co-constraint cannot be expressed with XSDs. Indeed, the counterexample is constructed from t_1 in Example 3.3 by replacing its first subtree by the first subtree of t_2.

6.4 Ancestor-based schemas

We introduce an instantiation of pattern-based schemas with the expressive power of single-type EDTDs. It is based on regular expressions. To this end, we associate with a regular expression r the pattern which selects those nodes v of a tree t for which anc-str$^t(v)$ satisfies r. The last two rules in the pattern-based schema of Example 5.1 thus become

$$\begin{array}{rcl} \Sigma^* \cdot \texttt{regulars} \cdot \texttt{dvd} & \to & \texttt{title} \\ \Sigma^* \cdot \texttt{discounts} \cdot \texttt{dvd} & \to & \texttt{discount} \end{array}$$

Here, Σ^* denotes the set of all Σ-strings. Next, we formally define such pattern-based schemas. Let t be a tree and v be a node with children $v_1, \ldots, v_n$, numbered from left to right. By ch-str$^t(v)$ we denote the string formed by the labels of the children of v, i.e., $\mathrm{lab}^t(v_1) \cdots \mathrm{lab}^t(v_n)$. Usually we omit the superscript t.

Definition 6.6. An **ancestor-based schema** S is a pattern-based schema (Σ, R), where all rules are of the form $r \to s$, where r and s are regular expressions over Σ. A tree t satisfies S if for every node v there is some $r \to s$ in R such that anc-str(v) matches r and ch-str(v) matches s.

We show in Theorem 7.1 below that the class of ancestor-based schemas corresponds *precisely* to the class of schemas represented by single-type EDTDs. In other words, the instantiation of the general framework introduced in Section 5 with regular expressions over ancestor-strings can be used as an alternative syntax for XML Schema. The underlying idea is the following: the type of any node when validated against a single-type EDTD depends uniquely on the type of its parent which in turn depends on the type of his parent and so on. These dependencies can be captured by an automaton and, hence, also by a regular expression over ancestor strings which leads to the formalism of ancestor-based schemas.

6.5 Ancestor-based patterns

The final characterization is based on the following notion:

Definition 6.7. Let T be a set of trees. We say that T **can be characterized by ancestor-based patterns** if there is a regular string language L over $\Sigma \cup \{\#\}$ such that, for every tree t, we have that $t \in T$ if and only if $P_{\mathrm{anc}}(t) \subseteq L$, where $P_{\mathrm{anc}}(t) = \{\text{anc-str}(v)\#\text{ch-str}(v) \mid v \in t\}$.

We show that ancestor-based schemas have ancestor-based patterns. Intuitively, each rule $r \to s$ in the ancestor-based schema corresponds to the regular expression $r \cdot \# \cdot s$. The regular language L then is the union of all these expressions.

PROPOSITION 6.8. *When a regular tree language T is definable by an ancestor-based schema then T can be characterized by ancestor-based patterns.*

PROOF. Let T be defined by the ancestor-based schema $S = (\Sigma, R)$. Then T can be characterized by the set $L = \{u\#v \mid u \in L(r), v \in L(s), r \to s \in R\}$. By definition, for every tree $t \in T$ it holds that $P_{\text{anc}}(t) \subseteq L$. For the other direction, let t be a tree which is not in T. Hence, there is a node w in t such that either there is no rule $r \to s$ in R with anc-str$(w) \in L(r)$ or for every such triple ch-str$(w) \notin L(s)$. This implies that anc-str$(w)\#$ch-str$(w) \notin L$. Therefore, a tree t is in T if and only if $P_{\text{anc}}(t) \subseteq L$ and we are done. □

In the next section, we show that an EDTD D is single-type iff $L(D)$ can be characterized by ancestor-based patterns. This characterization has interesting consequences for optimization. It shows that, just as for DTDs, equivalence and inclusion testing reduces to the corresponding problems on regular string languages as opposed to tree automata. We give more details in Section 10.3.

7. THE EQUIVALENCE THEOREM FOR ANCESTOR-BASED SCHEMAS

Before we state and prove our first technical result, we introduce some more notation.

We sometimes use a string notation for trees. That is, we denote by a the associated tree of $\langle a\rangle w\langle/a\rangle$, where w contains only text; and by $a(x'_1 \cdots x'_n)$, we denote the associated tree of $\langle a\rangle x_1 \cdots x_n\langle/a\rangle$, where each x'_i denotes the associated tree of document x_i. For each tree t, we define its set of nodes, Nodes(t), in a canonical way as follows. Every element in Nodes(t) is a sequence of natural numbers. The empty sequence ε represents the root of t. Furthermore, for any node u in t, its n children are represented by $u1, \ldots, un$ in the order given by the document (from left to right). By using this convention, a node u corresponds for every tree to the same position in the tree.

Given an extended DTD $D = (\Sigma, \Delta, d, s_d, \mu)$ and a type a^i, we denote by (D, a^i) the extended DTD D, where we replace s_d by a^i. We call D **trimmed** if d contains no unreachable rules, and for all $a^i \in \Delta$, $L((d, a^i)) \neq \emptyset$. Intuitively, D is trimmed if each of its types is assumed in at least one tree in $L(D)$. As reachability and testing emptiness of an EDTD is in PTIME [Martens and Neven 2005], an EDTD can be trimmed in PTIME. Thus, we will usually assume that EDTDs that are given as inputs are already trimmed.

As the structure of witness trees is the same as the structure of the trees they are witnesses of, we assume in proofs that they have the same set of nodes.

THEOREM 7.1. *For a homogeneous regular tree language T the following conditions are equivalent.*

(a) T is definable by a single-type EDTD.

(b) T is definable by an EDTD with ancestor-based types.

(c) T is closed under ancestor-guarded subtree exchange.

(d) T is definable by an ancestor-based schema.

(e) T can be characterized by ancestor-based patterns.

Proof. Note that (a)⇒(b)⇒(c) are already proved in Proposition 6.3 and Proposition 6.5. To close that group it suffices to show (c)⇒(a). Further (d)⇒(e) is proved in Proposition 6.8. It then suffices to show (a)⇒(d) and (e)⇒(b).

Let c denote the unique root label of the trees in T.

(a) ⇒ (d): Let T be defined by a single type EDTD $D = (\Sigma, \Delta, d, c^0, \mu)$ with $\cdot, \bot \notin \Delta$. Let A be a DFA over Σ with state set $Q = \Delta \cup \{\cdot, \bot\}$, initial state $\cdot$ and transition function $\delta : Q \times \Sigma \to Q$. Let $\delta(a^i, b)$ equal the unique b^j occurring in $d(a^i)$ if such a symbol exists, otherwise $\bot$. Furthermore, $\delta(\cdot, c) = c^0$. Note that the single-type property ensures that A is deterministic and well-defined.

Let $S = (\Sigma, R)$ be the ancestor based schema with rules of the form $r_{a,i} \to \mu(d(a^i))$, where $r_{a,i}$ is a regular expression describing the set $\{w \mid \delta^*(\cdot, w) = a^i\}$ of strings which bring A into state a^i. Of course, the languages $L(r_{a,1}), \ldots, L(r_{a,k_a})$ are all disjoint where $\{a^1, \ldots, a^{k_a}\}$ are the symbols mapped to a by μ. Note that we also denote by μ the homomorphic extension of μ to regular expressions $d(a^i)$.

It remains to show that S defines the same set of trees as D. Let t be in $L(D)$ and t' be a witness. It is easily shown by induction that, for each node v of t', $\mathrm{lab}^{t'}(v) = \delta^*(\cdot, \text{anc-str}^t(v))$. Hence, for each node v labeled with a^i, the rule of S responsible for v is $r_{a,i} \to \mu(d(a^i))$ and can therefore be applied. The proof of the opposite inclusion is similar.

(e) ⇒ (b): Let T be characterized by ancestor-based patterns using the language L. Let $A = (\Sigma, Q, \delta, s, F)$ be a DFA for L. Define $D = (\Sigma, \Delta, d, s_d, \mu)$ as follows. Δ is the set of all pairs (a, q), where $a \in \Sigma$ and $q \in Q$ and $\mu((a, q)) = a$. We let $d((a, q))$ be a regular expression describing all strings $(b_1, q_1) \cdots (b_n, q_n)$, for which A accepts $\#b_1 \cdots b_n$ when started from state q and $\delta(q, b_i) = q_i$, for every $i \leq n$. The start symbol s_d is (c, q') where $\delta(s, c) = q'$. By construction, D is single-type and therefore also has ancestor-based types. It is easy to see that $L(D)$ defines T. Indeed, when $t \in T$, let t' be obtained from t by relabeling every inner node v labeled a by (a, q) where $q = \delta^*(s, \text{anc-str}^t(v))$ then $t' \in L(D)$ and $t = \mu(t')$). Conversely, let $t' \in L(D)$. Then, for every node u of $t = \mu(t')$, $\text{anc-str}^t(u)\#\text{ch-str}^t(u) \in P_{\text{anc}}(t)$ by construction.

(c) ⇒ (a): Let $D = (\Sigma, \Delta, d, s_d, \mu)$ be an EDTD defining a tree language closed under ancestor-guarded subtree exchange. Our aim is to construct a single-type EDTD E such that $L(E) = L(D)$.

As explained in the beginning of this section, we assume without loss of generality that D only contains useful types, i.e., each type occurs in the witness of some tree in $L(D)$. For each type of D, choose a fixed tree, which is the subtree rooted at some node of this type in a tree in $L(D)$.

We will make use of the following general property of EDTDs:

(†) If t_1, t_2 are trees in $L(D)$ with witnesses t_1', t_2', respectively, such that v_1 in t_1 and v_2 in t_2 have the same type in t_1' and t_2', respectively, then the tree obtained from t_1 by replacing the subtree of v_1 with the subtree of v_2 in t_2 is in $L(D)$.

This property should not be confused with the subtree-exchange properties defined above which do not concern types at all.

For a string $w \in \Sigma^*$ and $a \in \Sigma$ let types(wa) be the set of all types a^i, for which there is a tree t with witness tree $t' \in L(d)$ and a node v in t such that anc-str$^t(v) = wa$ and the type of v in t' is a^i. For each $a \in \Sigma$, let $\tau(D, a)$ be the set of all nonempty sets types(wa), with $w \in \Sigma^*$. Clearly, each $\tau(D, a)$ is finite.

We next define $E = (\Sigma, \Delta_E, e, s_d, \mu_E)$. Its set of types is $\Delta_E := \bigcup_{a \in \Sigma} \tau(D, a)$. Note that $s_d \in \Delta_E$. For every $\tau \in \tau(D, a)$, set $\mu_E(\tau) = a$. In e, the right-hand side of the rule for each types(wa) is the disjunction of all $d(a^i)$ for $a^i \in$ types(wa), with each b^j in $d(a^i)$ replaced by types(wab). It should be noted that by (†), the definition of the rules of e does not depend on the actual choice of wa.

Clearly, E is single-type and $L(D) \subseteq L(E)$. Thus it only remains to show $L(E) \subseteq L(D)$.

To this end, let $g \in L(E)$ and let g' be a witness. We call a set S of nodes of g **well-formed** if (1) for each node $v \in S$ all its ancestors are in S and (2) if a child u of a node v is in S then all children of v are in S. The singleton set S_ε containing the root is well-formed.

We say that a tree t_2 **agrees** with a tree t_1 on an ancestor-closed set S_1 of nodes of t_1, if S_1 can be mapped to a well-formed S_2 by a mapping m which respects the child-relationship, the order of siblings and the labels of nodes.

As all trees in $L(D)$ and $L(E)$ have the same root label, there exists a tree $t_1 \in L(D)$ which agrees with g on S_ε. To complete the proof of "(c) ⇒ (a)" it is sufficient to prove the following.

Claim. If there exists a tree $t_1 \in L(D)$ which agrees with g on a well-formed set $S \subsetneq$ Nodes(t) then there exists $t_2 \in L(D)$ which agrees with g on a well-formed set which is a strict superset of S.

For the proof of this claim, let $wa =$ anc-str$^g(v)$, for some node $v \in S$ whose children are not in S. Let t_1 be as stated and let t_1' be its witness. Let a^i be the type of the node $m(v)$ corresponding to v in t_1'.

By construction of E the right-hand side of the rule for types(wa) is a disjunction over the (adapted) right-hand sides of rules of D. Let a^j be such that the children of v are typed in g' according to a disjunct derived from the rule for a^j. Thus, in particular, $a^j \in$ types(wa). Thus, there is a tree $t_3 \in L(D)$ with a node u such that anc-str$^{t_3}(u) = wa$ and the type of u is a^j in the witness t_3' for t_3.

Let, for each child v_1 of v in g, a type $f(v_1)$ be chosen such that ch-str(v) matches $d(a^j)$ with these types. Let t_4 be obtained from t_3 by (1) removing everything below u, (2) adding the children of v below u, and (3) adding for each child v_1 the fixed subtree chosen for $f(v_1)$. Clearly, by (†), $t_4 \in L(D)$. Furthermore, by the ancestor-closed subtree exchange property, the tree t_2 resulting from t_1 by replacing the subtree rooted at $m(v)$ by the subtree of t_4 rooted at u is in $L(D)$, too. This completes the proof of the claim and thus of "(c) ⇒ (a)".

■

8. TOWARDS A ROBUST NOTION OF TYPING

The rationale behind the Element Declarations Consistent constraint is that it allows for efficient and unique typing. Indeed, as discussed in Section 6.2, there is a simple one-pass top-down algorithm to validate and type a document against a schema. Although EDC is therefore clearly a sufficient condition for efficient

typing, the question arises to which extent the EDC constraint is also necessary. We consider here the requirement of efficient typing in a streaming fashion. Clearly, a document that can be typed in a top-down fashion can also be typed in a streaming manner, but the converse is not always true.

In this section, we provide several semantical and syntactical characterizations of this class of documents, as well as an instantiation of the pattern-based approach that also defines this class. These characterizations are all equivalent. As in Section 6, we provide the easy implication proofs in this section while the full equivalence proofs are delegated to Section 9. The section ends with a consideration of the Unique Particle Attribution rule (UPA). It is shown that UPA and EDC are incomparable, and that UPA like EDC, implies but is not equivalent to efficient typing in a streaming fashion.

8.1 1-pass preorder typing of XSDs

As mentioned before, the expressive power of EDTDs (and Relax NG) corresponds to the well-understood and very robust class of regular tree languages. However, this expressive power comes at a price. Although it can be determined in linear time whether a tree satisfies a given EDTD, the way to do that is sometimes at odds with the way one would like to process XML documents. More concretely, it requires to process documents in a bottom-up fashion where the type(s) of an element is only determined after reading its content. In the context of streaming XML data or for SAX-based processing, i.e., when processing an XML document as a SAX-stream of opening and closing tags, it is more desirable to determine the type of an element at the time its opening tag is met. If an EDTD fulfills this requirement we say it is *1-pass preorder typeable* (1PPT). Note that not every EDTD admits 1PPT. Consider the example $a \to b^1 + b^2$, $b^1 \to c$, $b^2 \to d$ and the document `<a><b><d/></b></a>`. The type of `b` depends on the label of its child. It is hence impossible to assign a type to `b` when its opening tag `<b>` is met, i.e., without looking at its child. An alternative formulation of 1PPT is that the type of an element cannot depend on anything occurring in document order after the opening tag of that element. Hence, we require that a type is uniquely determined by the preceding of an element (cf. Figure 6). On top of one-pass preorder typeability, this notion therefore also enforces the attribution of a unique type to every element. The latter is, for instance, not the case for Relax NG which allows ambiguous typing as in the grammar $a \to b^1 + b^2$, $b^1 \to c$, and $b^2 \to c$, where b can both be assigned type b^1 and b^2 in the tree $a(b(c))$.

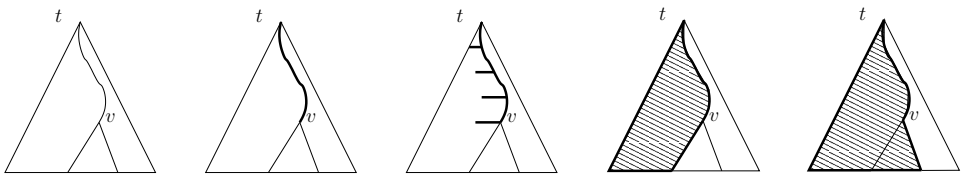


Fig. 6. From left to right: a tree t, the ancestor-string of v, the ancestor-sibling-string of v, the preceding of v and the preceding-subtree of v in t.

In the XML Schema specification as well as in research papers, various kinds of constraints have been defined that enable efficient validation and typing of XML documents. Although it is hardly made precise what efficient typing should mean exactly, one might argue that the intention roughly matches our notion of 1-pass preorder typeability. It should be noted here that 1PPT is a semantical notion, while the proposed notion of single-type EDTDs, for instance, is a syntactic one as its definition refers to syntactic restrictions of the schema rather than to the documents themselves. However, 1PPT is a robust notion precisely because it is semantic: it defines the largest class of EDTDs that can be typed when processed in a streaming fashion.

We formalize the notion of 1PPT in terms of preceding-based types in analogy to the ancestor-based types of Definition 6.2. The **preceding** of a node v in t is the tree resulting from t by removing everything below v, all right siblings of v's ancestors and of v, and their respective subtrees (cf. Figure 6). In other words, the preceding of v in t is the subtree of t consisting of all nodes that are before v in document order, and v itself. We denote the preceding of v by $\text{preceding}^t(v)$.

Definition 8.1. We say that an EDTD $D = (\Sigma, \Delta, d, s_d, \mu)$ **is 1-pass preorder typeable** (1PPT) or **has preceding-based types** if there is a function $f : \mathcal{T}_\Sigma \to \Delta$ such that, for each tree $t \in L(D)$,

—there is exactly one witness t', and

—t' results from t by assigning to each node v the type $f(\text{preceding}^t(v))$.

Theorem 7.1 characterizes single-type EDTDs precisely as the class of EDTDs with ancestor-based types. Therefore, every single-type EDTD admits 1PPT. The converse, however, is not true. Consider for example the following EDTD which is not single-type: $a \to b^1\, b^2$, $b^1 \to c$, $b^2 \to d$. Nevertheless, the EDTD admits 1PPT. Indeed, it is easy to see that the EDTD only defines the singleton `<a><b><c/></b><b><d/></b></a>`. The rule for a says that the first b-child needs to be typed b^1 and the second b-child needs to be typed b^2. For each of the b's in the document, it can be easily determined whether it is the first or second child of a by investigating its preceding (cf. Figure 6). Hence, the notion of single-type EDTDs allows for efficient unique typing, but does not capture all of 1PPT EDTDs.

8.2 Ancestor-sibling-based types

One of the more surprising results of this paper is that although the definition of 1PPT explicitly allows dependence on the complete preceding for the type of an element, in the context of EDTDs, already dependence on the ancestor-sibling-string (as defined next) suffices.

Let t be a tree, v a node in t and $u_1, \ldots, u_k$ its left siblings. By $\text{l-sib-str}^t(v)$, we denote the string $\text{lab}^t(u_1) \cdots \text{lab}^t(u_k)\text{lab}^t(v)$. The **ancestor-sibling-string of** v, denoted by $\text{anc-sib-str}^t(v)$, is the string

$$\text{l-sib-str}^t(v_1)\#\text{l-sib-str}^t(v_2)\# \cdots \#\text{l-sib-str}^t(v_n)\#\text{l-sib-str}^t(v)$$

formed by concatenating the left-sibling strings of all ancestors $v_1, v_2, \ldots, v_n$ of v starting from the root v_1 (c.f. Figure 6).

Definition 8.2. We say that an extended DTD $D = (\Sigma, \Delta, d, s_d, \mu)$ **has ancestor-sibling-based types** if there is a function $f : (\Sigma \cup \{\#\})^* \to \Delta$ such that, for each tree $t \in L(D)$,

—there is exactly one witness t', and

—t' results from t by assigning to each node v the type $f(\text{anc-sib-str}^t(v))$.

The next proposition immediately follows by definition:

PROPOSITION 8.3. *When an EDTD has ancestor-sibling-based types, it also has preceding-based types.*

8.3 A characterization of 1PPT by a subtree-exchange property

Just as for single-type EDTDs, EDTDs admitting 1PPT satisfy a very simple closure property which provides a means to prove that certain tree languages are not definable by 1PPT EDTD.

Recall that $t_1[u \leftarrow t_2]$ denotes the tree obtained from a tree t_1 by replacing the subtree rooted at node u of t_1 by t_2, and that $\text{subtree}^t(u)$ denotes the subtree of t rooted at u.

Definition 8.4. A tree language T is **closed under ancestor-sibling-guarded subtree exchange** if the following holds. Whenever for two trees $t_1, t_2 \in T$ with nodes u_1 and u_2, respectively, $\text{anc-sib-str}^{t_1}(u_1) = \text{anc-sib-str}^{t_2}(u_2)$ then $t_1[u_1 \leftarrow \text{subtree}^{t_2}(u_2)] \in T$.

The definition is illustrated by Figure 7.

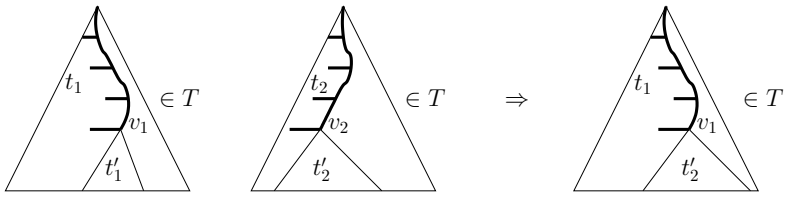


Fig. 7. Ancestor-sibling-guarded subtree exchange.

Note that the above notions extend the notion of ancestor-based types of Definition 6.2. The proof of the following proposition is then also almost identical to the one of Proposition 6.5, just replace ancestor by ancestor-sibling.

PROPOSITION 8.5. *When a regular tree language T has ancestor-sibling-based types then T is closed under ancestor-sibling-guarded subtree exchange.*

As an immediate consequence, the language we considered in Example 3.3 is not definable by an EDTD admitting 1PPT. Note that the counterexample can be constructed in exactly the same manner.

8.4 Restrained competition EDTDs

We recall the definition of *restrained competition* EDTDs introduced by Murata, Lee, and Mani [Murata et al. 2005]. They can be seen as a generalization of single-type EDTDs.

Definition 8.6. Let $D = (\Sigma, \Delta, d, s_d, \mu)$ be an EDTD. A regular expression r (over the alphabet of types Δ) **restrains competition** if there are no strings $w\tau v$ and $w\tau' v'$ in $L(r)$ with $\tau \neq \tau'$ and $\mu(\tau) = \mu(\tau')$. The EDTD D is **restrained competition** iff all regular expressions occurring in rules restrain competition.

Intuitively, a restrained competition regular expression ensures that when visiting the children of a node from left to right it is always clear which type is associated to each node without seeing its right siblings. So, single-type implies restrained competition.

Example 8.7. The following is an example of a restrained competition EDTD that is not single-type nor has an equivalent single-type EDTD.

$$\begin{array}{lll} \texttt{store} & \rightarrow & (\texttt{reg-dvd})^* \ \texttt{discounts} \ (\texttt{dis-dvd})^* \\ \texttt{discounts} & \rightarrow & \varepsilon \\ \texttt{reg-dvd} & \rightarrow & \texttt{title price} \\ \texttt{dis-dvd} & \rightarrow & \texttt{title price discount} \end{array}$$

where $\mu(\texttt{reg-dvd}) = \mu(\texttt{dis-dvd}) = \texttt{dvd}$. The expression

$$(\texttt{reg-dvd})^* \ \texttt{discounts} \ (\texttt{dis-dvd})^*$$

is restrained competition as types can be assigned from left to right: each time a `dvd`-element is read, it has type `reg-dvd` when `discounts` has not been met yet, and type `dis-dvd`, otherwise.

In contrast, the expression $(\texttt{reg-dvd} + \texttt{dis-dvd})^*\texttt{dis-dvd}(\texttt{reg-dvd} + \texttt{dis-dvd})^*$ of Example 2.4 is not restrained competition as the strings `dis-dvd` and `reg-dvd dis-dvd` are both defined by the regular expression but `reg-dvd` and `dis-dvd` are associated to the same element name. Here, $w = \varepsilon$, $\tau = \texttt{dis-dvd}$, $\tau' = \texttt{reg-dvd}$, $v = \varepsilon$, and $v' = \texttt{dis-dvd}$. □

We show in Theorem 9.1, that any homogeneous regular tree language that admits 1PPT, can be defined by a restrained competition EDTD.

This restriction allows a strictly larger class of schemas than EDC while still permitting a unique top-down left-to-right assignment of types as discussed in Section 8.2. Note that both the single-type and the restrained competition constraint are local: they restrain the structure of admissible regular expressions. Unfortunately, EDC is syntactic while restrained competition is a semantical notion. Nevertheless, whether an EDTD is restrained competition can be decided in polynomial time (Section 10, Theorem 10.2).

8.5 Ancestor-sibling-based schemas

To raise the expressiveness of pattern-based schemas to the level of EDTDs admitting 1PPT, we need an adequate pattern language. To this end, we use a set $\mathcal{R}$ of regular expressions over symbols $a[r]$ where r is a regular expression over element names and a is an element name. We simply write a for $a[\Sigma^*]$. For instance,

$(a[a+b^*]+b)^*a[b^*]$ is an expression of $\mathcal{R}$ with three symbols, $a[a+b^*]$, $b[\Sigma^*]$ and $a[b^*]$. We say that $a[r]$ **matches node** v when v is labeled with a and the string formed by the labels of the left siblings of v match r.

We explain how an expression φ can be used as a unary pattern. Let v be a node of a tree t. Let $v_1, \ldots, v_n$ be the path from the root v_1 to $v = v_n$. For each i, let a_i denote the label of v_i and let w_i be the string of labels of the left siblings of v_i, without the label of v_i itself. Node v is **selected by pattern** φ iff there exists a string $a_1 a_2[r_2] \cdots a_n[r_n] \in L(\varphi)$ such that for every $i = 2, \ldots, n$, $w_i \in L(r_i)$. In other words, for each symbol $a_i[r_i]$, r_i constrains the left siblings of the node v_i.

Example 8.8. Using $\mathcal{R}$ as pattern language, we can define the EDTD of Example 8.7 in our framework in the following way:

$$\begin{aligned} \texttt{store} &\to \texttt{dvd}^* \ \texttt{discounts} \ \texttt{dvd}^* \\ \texttt{discounts} &\to \varepsilon \\ \texttt{store dvd}[\texttt{dvd}^*] &\to \texttt{title price} \\ \texttt{store dvd}[\texttt{dvd}^* \ \texttt{discounts} \ \texttt{dvd}^*] &\to \texttt{title price discount} \end{aligned}$$

□

For a formal definition, we recall that ch-str$^t(v)$ denotes the string formed by the labels of the children of a node v.

Definition 8.9. An **ancestor-sibling-based schema** S is a pattern-based schema (Σ, R), where all rules are of the form $\varphi \to s$, where s is a regular expression over Σ and φ is a pattern from $\mathcal{R}$. A tree t satisfies S if for every node v there is some $\varphi \to s$ in R such that v is selected by pattern φ and ch-str(v) matches s.

By Theorem 9.1 below, ancestor-sibling-based schemas correspond precisely to the class of schemas represented by restrained competition EDTDs and therefore to those EDTDs admitting 1PPT. In other words, the instantiation of the general framework with regular expressions over ancestor-sibling-strings is an alternative syntax for *all* EDTDs admitting 1PPT.

As in Subsection 5.3, we can adopt two approaches to employ schemas for practical settings: enhance DTDs or extend XML Schema. To capture 1PPT EDTDs it suffices to add $\mathcal{R}$-patterns. A more practical way is to add full XPath, but semantically restrict its evaluation to the preceding of each node (cf. Figure 6). For instance, the expression `//*[.//b]//c` selects only those `c`-elements having a `b`-element in their preceding as illustrated in Figure 8.

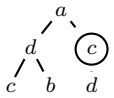


Fig. 8. Only the circled `c`-element in the document has a `b`-element in its preceding.

8.6 Ancestor-sibling-based patterns

The following definition characterizes EDTDs admitting 1PPT in terms of allowable patterns.

Definition 8.10. Let T be a set of trees. We say that T **can be characterized by ancestor-sibling-based patterns**, if there is a regular string language L such that, for every tree t, we have that $t \in T$ if and only if $P_{\text{anc-sib}}(t) \subseteq L$, where $P_{\text{anc-sib}}(t) = \{\text{anc-sib-str}(v)\#\text{ch-str}(v) \mid v \in t\}$.

The proof of the next Proposition is similar to the one of Proposition 6.8 and is therefore omitted.

PROPOSITION 8.11. *When a regular tree language T is definable by an ancestor-sibling-based schema, then T can be characterized by ancestor-sibling-based patterns.*

In the next section, we show that an EDTD D is 1PPT iff $L(D)$ can be characterized by ancestor-sibling-based patterns. This characterization is mostly relevant for optimization problems. It shows that, just as for DTDs and single-type EDTDs, equivalence and inclusion testing reduces to the corresponding problems on regular string languages. We provide more details in Section 10.3.

8.7 Unique Particle Attribution Rule

The most well-known XML Schema constraint is perhaps the Unique Particle Attribution (UPA) rule. In [van der Vlist 2002], it is mentioned that EDC and UPA are interrelated, in the sense that when a schema satisfies one constraint it almost always also satisfies the other. Although this might be true on most practical examples, in general it is definitely not the case. As we now show, the constraints are incomparable: they are related only in the weak sense that each of them alone implies 1PPT.

An EDTD satisfies the UPA constraint when, for every regular expression r over the type alphabet Δ, the expression $\mu(r)$, obtained from r by replacing every type τ by the element $\mu(\tau)$, is one-unambiguous (cf. Definition 3.1). The expression $a^1(a^2+b^1)$, for instance, is not EDC but satisfies UPA. For the other counter example, consider the expression $r = (a^1 + b^1)^*a^1(a^1 + b^1)$ which is clearly EDC. When matching a string against this expression, we always know that we need to type a and b by a^1 and b^1, respectively. However, the expression $\mu(r) = (a+b)^*a(a+b)$ is *not* one-unambiguous. Indeed, $a_1a_2a_3$ and a_2a_3 are both in $L((a_1+b_1)^*a_2(a_3+b_2))$. In [Brüggemann-Klein and Wood 1998] it is even shown that $\mu(r)$ can not be defined by *any* one-unambiguous regular expression. So, none of the EDC or UPA constraints implies the other.

The definition of UPA and restrained competition regular expressions are related in the following way. When matching a string against a restrained competition regular expression the type of the next element only depends on the part of the string already seen. For a one-unambiguous regular expression over the type alphabet as defined in the previous paragraph, the symbol in the regular expression that matches the next input element only depends on the part of the string already seen. As the matched symbol in the regular expression is actually the type of that symbol, it is immediate that every such one-unambiguous regular expression is restrained competition and, therefore, UPA implies 1PPT.

Example 8.12. Suppose that $r = a^1?b^1(b^1+c^1)^*a^2c^1$. Then, $\mu(r) = a?b(b+c)^*ac$ and $\overline{\mu(r)} = a_1?b_1(b_2+c_1)^*a_2c_2$. Clearly, $\mu(r)$ is one-unambiguous, which means that when we match e.g. *bbcbac* against $\mu(r)$, the symbol against which the a must be

matched (a_2 in $\overline{\mu(r)}$), is uniquely determined without looking ahead. But then, the symbol in r that corresponds to a^2 is also uniquely determined, and this symbol has only one type. So, we also know what type must be assigned to a without looking ahead to c. It is easy to generalize this example to show that any EDTD satisfying UPA is also restrained competition and therefore implies 1PPT.

Although the XML Schema specification allows typing in multiple passes (Section 5.2 in [Thompson et al. 2004], note on multiple assessment episodes), the previous discussion shows that already the EDC or UPA alone allow for one-pass typing (as they imply 1PPT). Nevertheless, neither EDC nor UPA captures the class of all 1PPT schemas.

There has been quite some debate in the XML community about the restriction to 1-unambiguous regular expressions (cf., e.g., pg 98 of [van der Vlist 2002] and [Mani 2001; Sperberg-McQueen 2003]) as it does not serve its purpose: even for general regular expressions simple validation algorithms exist that are as efficient as those for one-unambiguous regular expressions. One reason to maintain this restriction is to ensure compatibility with SGML parsers, the predecessor of XML. The results of this paper show that, on the other hand, by using restrained competition EDTDs instead, a larger expressive power can be achieved without (essential) loss in efficiency. For both classes, validation and typing is possible in linear time, allowed schemas can still be recognized in quadratic time and an allowed schema can be constructed in exponential time, if one exists [Brüggemann-Klein and Wood 1998] (cf. Section 10).

On the negative side, both 1-unambiguous expressions and restrained competition expressions lack a comprehensive syntactical counterpart. Whether such an equivalent syntactical restriction exists remains open. It would also be interesting to find syntactic restrictions which imply an efficient construction of an equivalent restrained competition EDTD.

9. THE EQUIVALENCE THEOREM FOR 1-PASS PREORDER TYPEABLE SCHEMAS

In this section, we prove the following theorem.

THEOREM 9.1. *For a homogeneous regular tree language T the following conditions are equivalent.*

(a) T is definable by a 1-pass preorder typeable EDTD.

(b) T is definable by a restrained competition EDTD.

(c) T is definable by an EDTD with ancestor-sibling-based types.

(d) T is closed under ancestor-sibling-guarded subtree exchange.

(e) T can be characterized by ancestor-sibling-based patterns.

(f) T is definable by an ancestor-sibling-based schema.

Proof. We show (a) $\Leftrightarrow$ (c) and (c) $\Rightarrow$ (d) $\Rightarrow$ (b) $\Rightarrow$ (f) $\Rightarrow$ (e) $\Rightarrow$ (c). Of these (c) $\Rightarrow$ (a), (c) $\Rightarrow$ (d), and (f)$\Rightarrow$ (e) are stated in Proposition 8.3, Proposition 8.5, and Proposition 8.11, respectively. Further, (e) $\Rightarrow$ (c) is a straightforward generalization of the proof of (e) $\Rightarrow$ (b) in Theorem 7.1.

(b) $\Rightarrow$ (f):

Let T be defined by a restrained competition EDTD $D = (\Sigma, \Delta, d, s_d, \mu)$. We are going to construct a DFA A which determines the type of a node v, after reading its ancestor-sibling-string. From this DFA, we will then obtain an ancestor-sibling-based schema.

For each symbol a^i in Δ, let $A_{a,i} = (Q_{a,i}, \Delta, \delta_{a,i}, s_{a,i}, F_{a,i})$ be a minimal DFA for $L(d(a^i))$. We require that the sets $Q_{a,i}$ are pairwise disjoint. Because it is minimal, each $A_{a,i}$ has at most one state $q^\perp$ from which no accepting state is reachable and it has no unreachable states. From the restrained competition property it immediately follows that, for each state q of $A_{a,i}$, if $\delta(q, b^j) = q_1$, $\delta(q, b^k) = q_2$, $q_1 \neq q_2$ and $j \neq k$ then q_1 or q_2 must be $q^\perp$.

The desired DFA $A = (Q_A, \Sigma, s_A, \delta_A, F_A)$ is constructed as follows. The set Q_A consists of all pairs (q, b), where $q \in Q_{a,i}$, for some a^i, and $b \in \Delta \cup \{\#\}$. Intuitively, q is the current state of an automaton $A_{a,i}$ and b is the last type that has been identified. If $s_d = a^\ell$, the initial state s_A of A is $(s_{a,\ell}, \#)$. The transition function δ_A is defined as follows. For each $q \in Q_{a,i}$, $c \in \Delta \cup \{\#\}$ and $b \in \Sigma$ we let $\delta_A((q, c), b) = (\delta_{a,i}(q, b^j), b^j)$, for the unique j with $\delta_{a,i}(q, b^j) \neq q^\perp$, if such a j exists. Otherwise, $\delta_A((q, c), b) = (q^\perp, \#)$. Furthermore, we let $\delta_A((q, b^j), \#) = (s_{b,j}, \#)$. The set F_A can be chosen arbitrarily, as we do not make use of final states.

From the definition, it is obvious that, for each node v of a tree in T,

$$\delta_A^*(s_A, \text{anc-sib-str}(v)) = (q, a^i),$$

for some q, where a^i is the unique type of v.

Now we are ready to define the ancestor-sibling-based schema S. For each state (q, a^i) of A, let L_{q,a^i} denote $\{w \mid \delta_A^*(s_A, w) = (q, a^i), \text{ for some } q\}$. It is possible to construct a regular language R_{q,a^i} (and therefore, also a regular expression r_{q,a^i}) over expressions of the form $a[r]$ such that a string $a_1 \# w_2 a_2 \# \cdots \# w_k a_k$ is in L_{q,a^i} if and only if there is a string $a_1 a_2[r_2] \cdots a_k[r_k]$ in $L(r_{q,a^i})$ and, for each j, $w_j \in L(r_j)$. Indeed, let $r_{q,q'}$ be a regular expression defining all strings w without a separator $\#$ that take A from state q to state q'. Then the alphabet of R_{q,a^i}, consists of all symbols $b[r_{q,q'}]$. As L_{q,a^i} can be mapped onto R_{q,a^i} by a generalized sequential machine, the latter is a regular set (cf., e.g., [Hopcroft and Ullman 1979]). Then, S consists of all rules $r_{q,a^j} \to \mu(d(a^j))$. Note, that the languages $L(r_{q,a^i})$ are pairwise disjoint by construction.

It remains to show that S and D describe the same tree language.

To this end, let first $t \in L(D)$ and let v be a node of t. Let (q, a^i) be the state of A after reading anc-sib-str(v). Thus, in the unique labeling of t with respect to D, v has type a^i. Hence, ch-str(v) is in $\mu(d(a^i))$ and $r \to s$ is fulfilled at v.

For the converse direction, let $t \in L(S)$ and let v be a node of t. Let $r \to s$ be the unique rule for which anc-sib-str(v) matches r. By construction, $r \to s$ corresponds to a type a^i for which $\delta_A^*(s_A, \text{anc-sib-str}(v)) = (q, a^i)$. In this way, a unique labeling of t by types is induced and it is straightforward that this labeling is valid with respect to D.

(a) ⇒ (c):

We show even a bit more than required: each EDTD with preceding-based types *already has* ancestor-sibling based types.

Let $D = (\Sigma, \Delta, d, s_d, \mu)$ be an EDTD which has preceding-based types. Towards a contradiction, we assume that D has types which are *not* ancestor-sibling based.

Clearly, because D has preceding-based types, the types of each $t \in L(D)$ are uniquely determined, thus, only the second requirement of Definition 8.2 can fail. Hence, there are trees $t_1, t_2 \in L(D)$ with nodes v_1 in t_1 and v_2 in t_2 such that anc-sib-str$^{t_1}(v_1)$ = anc-sib-str$^{t_2}(v_2)$ but v_1 has a different label in t_1' than v_2 in t_2', where t_1' and t_2' are the unique witnesses for t_1 and t_2, respectively. We call t_1, t_2, v_1, v_2 a *counterexample*. Let t_1, t_2, v_1, v_2 be a counterexample for which the length of anc-sib-str$^{t_1}(v_1)$ is minimal.

Let U_1 be the set of nodes which are left siblings of ancestors of v_1 let U_2 be the corresponding set for v_2. As anc-sib-str$^{t_1}(v_1)$ = anc-sib-str$^{t_2}(v_2)$, there is a natural bijection f from U_1 to U_2. Clearly, for each $v \in U_1$, v and $f(v)$ have the same label.

Let s be the tree resulting from t_1 by replacing each node $v \in U_1$ and its subtree by $f(v)$ and its subtree. As the counterexample was chosen minimally, for each $v \in U_1$, the label of v in t_1' is the same as the label of $f(v)$ in t_2'. Let s' be the tree resulting from s by labelling each subtree of a node $v \in f(U_1)$ as in t_2' and all other nodes as in t_1'.

It is easy to see that $s' \in L(d)$. As preceding$^{s}(v_1)$ = preceding$^{t_2}(v_2)$, and as we assume preceding-based types, v_1 must have the same label in s' as v_2 in t_2'. As it also has the same label in t_1' as in s' it follows that the labels in t_1' and t_2' are the same which leads to the desired contradiction.

(d) ⇒ (b): The proof is similar to but a bit more involved than the corresponding proof "(c) ⇒ (a)" in Theorem 7.1.

Let $D = (\Sigma, \Delta, d, s_d, \mu)$ be an EDTD defining a tree language closed under ancestor-sibling-guarded subtree exchange. We will construct a restrained competition EDTD $E = (\Sigma, \Delta_E, e, s_d, \mu_E)$ such that $L(E) = L(D)$. Again, we assume without loss of generality that D only contains useful types.

For a string $w \in (\Sigma \cup \{\#\})^*$ and $a \in \Sigma$ let types(wa) be the set of all types a^i, for which there is a tree t with witness tree $t' \in L(d)$ and a node v in t such that anc-sib-str$^{t}(v) = wa$ and the type of v in t' is a^i. For each $a \in \Sigma$, let $\tau(D, a)$ be the set of all nonempty sets types(wa), with $w \in (\Sigma \cup \{\#\})^*$. Again, each $\tau(D, a)$ is finite. The set of types of E is $\Delta_E := \bigcup_{a \in \Sigma} \tau(D, a)$ and, for each $\tau \in \tau(D, a)$, $\mu_E(\tau) = a$.

To define e, let $C \in \Delta_E$ and let $C = \{a^1, \ldots, a^\ell\}$ = types(wa) for a string wa. Then define L_C as the following regular language over Δ_E. It consists of all Δ_E-strings $x = x_1 \ldots x_n$ for which there is an $i \leq \ell$ and a string $x' \in L(d(a^i))$, such that $\mu(x') = \mu_E(x)$ and the j-th position of x is types$(wa\#\mu_E(x_1 \ldots x_j))$. Note that L_C does not depend on the choice of wa.

Intuitively, L_C is the union of all $d(a^i)$ where every jth Σ-symbol in a string $y_1 \cdots y_n$ is assigned the set of types types$(wa\#y_1 \ldots y_j)$. It should be clear that L_C is indeed restrained competition.

We next show that L_C is regular. We define an NFA M_C accepting L_C of size exponential in the size of D. To this end, let for each type a^i, $A_{a,i} = (\Delta, Q_{a,i}, \delta_{a,i}, s_{a,i}, F_{a,i})$ be an NFA for $L(d(a^i))$. W.l.o.g., we assume that the sets $Q_{a,i}$ are pairwise disjoint and that from every state in $Q_{a,i}$, a final state is reachable.

Define $M_C = (\Delta_E, Q_C, \delta_C, s_C, F_C)$ as follows:

—$Q_C = 2^Q_{a,1} \times \cdots \times 2^Q_{a,\ell}$;
—$s_C = (\{s_{a,1}\}, \ldots, \{s_{a,\ell}\})$;

—$F_C = \{(P_1, \ldots, P_\ell) \in Q_C \mid \exists i, P_i \cap F_{a,i} \neq \emptyset\}$;

—In order to define $\delta_{a,M}$, let $\overline{P} = (P_1, \ldots, P_\ell)$ be a state of M_C. Then, each P_i contains precisely the states in which each $A_{a,i}$ is after reading the input so far. For a state q of $A_{a,i}$ and a Σ-symbol b, let $\text{types}_{a,i}(q, b)$ consist of those types b^j for which $\delta_{a,i}(q, b^j) \neq \emptyset$. For a set P of states of $A_{a,i}$, define

$$\text{types}_{a,i}(P, b) = \bigcup_{q \in P} \text{types}_{a,i}(q, b).$$

Finally, for $\overline{P}$ as above, define $\text{types}_{a,i}(\overline{P}, b) = \bigcup_{j=1}^{\ell} \text{types}_{a,i}(P_j, b)$. Notice that, when starting from the state $\overline{P}$, for each $b \in \Sigma$, M_C can only make a transition when reading the Δ_E-symbol $\text{types}_{a,i}(\overline{P}, b)$. Therefore, $\delta_C(\overline{P}, \text{types}_{a,i}(\overline{P}, b)) = (P'_1, \ldots, P'_\ell)$ where $P'_i = \bigcup_j \delta_{a,i}(q, b^j)$. For all other $C' \in \Delta_E$ with $C' \neq \text{types}_{a,i}(\overline{P}, b)$, set $\delta_C(\overline{P}, C') = \emptyset$.

Note that $L_C = L(M_C)$. Indeed, M_C simulates every $A_{a,i}$ in parallel while computing $\text{types}(wa\#y_1 \ldots y_j)$ for every jth symbol in the Σ-string $y_1 \ldots y_n$ from left to right.

Let t be a tree in $L(D)$ witnessed by t'. It is not hard to show by proceeding from the root to the leaves that t' can be transformed to a tree t'' witnessing that $t \in L(E)$. The crucial point is, that the type of each node v in t' is an element of its type in t''.

Thus, it only remains to show $L(E) \subseteq L(D)$. This proof is completely analogous to the corresponding proof in Theorem 7.1. Only the notions depending on ancestors now depend on the corresponding notions for ancestors and their siblings. ∎

Theorem 9.1 shows that in the context of EDTDs having preceding-based types implies ancestor-based types. From the proof it further follows that for each such language a very simple and efficient typing algorithm exists. It is basically a deterministic pushdown automaton with a stack the height of which is bounded by the depth of the document. For each opening tag it pushes one symbol, for each closing tag it pops one. Hence, it only needs a constant number of steps per input symbol. In particular, it works in linear time in the size of the document. It should be noted that such automata have been studied in [Segoufin and Vianu 2002] and [Koch and Scherzinger 2003] in the context of streaming XML documents. The subclass of the context-free languages accepted by such automata has recently been studied in [Alur and Madhusudan 2004]. Thus, just like for single-type EDTDs, there is an efficient one-pass validation and typing algorithm.

10. STATIC ANALYSIS AND OPTIMIZATION

In this section, we consider various decision problems that are important for any automated treatment of schemas. In particular, we consider the following problems:

Recognition: Given an EDTD, check whether it is of a restricted type, i.e., a DTD, a single-type EDTD or a restrained competition EDTD.

SIMPLIFICATION: Given an EDTD, check whether it has an equivalent EDTD of a restricted type, i.e., an equivalent DTD, single-type EDTD or restrained competition EDTD.

CONTAINMENT: Given two EDTDs D_1, D_2, does D_1 describe a sublanguage of D_2?

Note the difference between RECOGNITION and SIMPLIFICATION. The former checks whether a given EDTD *is* of a specific form, while the latter checks whether the tree language defined by the given, possibly unrestricted, EDTD can be defined by a constrained EDTD. For instance, the non single-type EDTD $a \to b^1 b^2$, $b^1 \to c$, $b^2 \to c$ is clearly equivalent to the DTD consisting of the rules $a \to bb$ and $b \to c$.

The proofs in this section make use of tree automata for unranked trees. We recall the necessary definitions next. The robust notions of regular languages of strings and ranked trees can easily be generalized to unranked trees. The latter class is usually defined in terms of non-deterministic tree automata and possesses similar closure properties [Brüggemann-Klein et al. 2001]. We refer the unfamiliar reader to [Neven 2002b].

Definition 10.1. A **nondeterministic tree automaton (NTA)** is a tuple $B = (Q, \Sigma, \delta, F)$, where Q is a finite set of states, $F \subseteq Q$ is the set of final states, and δ is a function $\delta : Q \times \Sigma \to 2^{Q^*}$ such that $\delta(q, a)$ is a regular string language over Q for every $a \in \Sigma$ and $q \in Q$.

A **run** of B on a tree t is a labeling $\lambda : \text{Nodes}(t) \to Q$ such that for every $v \in \text{Nodes}(t)$ with n children $v_1, \ldots, v_n$ from left to right, $\lambda(v_1) \cdots \lambda(v_n) \in \delta(\lambda(v), \text{lab}^t(v))$. Note that when v has no children, then the criterion reduces to $\varepsilon \in \delta(\lambda(v), \text{lab}^t(v))$. A run is **accepting** iff the root is labeled with an accepting state. Note that a run can be seen as a bottom-up labeling of the input tree which accepts if a final state is assigned to the root. A tree is **accepted** if there is an accepting run. The set of all accepted trees is denoted by $L(B)$. The class of tree languages accepted by NTAs is called the **unranked regular tree languages**.

An NTA is **bottom-up deterministic** iff $\delta(q, a) \cap \delta(q', a) = \emptyset$ for all $q \neq q'$.

10.1 Recognition of EDTDs

We first consider the RECOGNITION problem. As the definition of a DTD and single-type EDTD is syntactical in nature, it can be immediately verified by an inspection of the rules whether an EDTD is in fact a DTD or a single-type EDTD. The case of restrained competition EDTDs is considered in the following Theorem.

THEOREM 10.2. *Given an EDTD D, there is an algorithm that tests whether D is restrained competition in time quadratic in the size of D.*

Proof. It suffices to show that testing whether a single regular expression is restrained competition can be done in quadratic time. Therefore, let r be a regular expression, and let $N_r = (\Delta, Q, \delta, q_0, F)$ be an NFA equivalent to r. The latter can be constructed in time $\mathcal{O}(n \log^2(n))$ resulting in $\mathcal{O}(n)$ states where n is the size of r [Hromkovic et al. 2001].

The algorithm makes use of two sets:

—the set of reachable states $R := \{q \in Q \mid \exists w \in \Delta^*, \delta^*(q, w) \in F\}$; and,

—the set of pairs of states that can be reached by the same string, $S := \{(q_1, q_2) \in Q \times Q \mid \exists w \in \Delta^*, \{q_1, q_2\} \subseteq \delta^*(q_0, w)\}$.

Note that R and S can be computed in linear and quadratic time, respectively, by the usual reachability algorithm. Then, r is restrained competition iff there are no $(q_1, q_2) \in S$ and a, i, j with $i \neq j$, $\delta(q_1, a^i) \cap R \neq \emptyset$ and $\delta(q_2, a^j) \cap R \neq \emptyset$. Altogether, a careful implementation leads to a quadratic time algorithm. ∎

We note that the above construction can be carried out in NLOGSPACE as well.

10.2 Simplification of EDTDs

Next, we study the complexity of the SIMPLIFICATION problem for the target schema types DTD, single-type EDTD and restrained competition EDTD, respectively. Unfortunately, this test is complete for exponential time. Our algorithm also constructs a corresponding equivalent simpler schema when it exists.

THEOREM 10.3. *Each of deciding whether an EDTD has an equivalent DTD, single-type EDTD or restrained competition EDTD is* EXPTIME*-complete.*

Proof. We start with the lower bounds. In all three cases, the lower bound is obtained by a reduction from the universality problem for non-deterministic tree automata [Seidl 1990]. Let NTA(REG) denote the class of NTAs where the regular languages encoding the transition function are represented by regular expressions. The hardness result even holds for NTA(REG) where automata only have one final state and where all accepted trees have the same root symbol (say a).

Therefore, let $A = (Q, \Sigma, \delta, F)$ be an NTA(REG) over alphabet $\Sigma = \{a, b\}$ with one final state $F = \{q_F\}$. We can assume w.l.o.g. that A accepts trees of depth at least two. We can construct in LOGSPACE an equivalent EDTD $D = (\Sigma, \Delta, d, a^{q_F}, \mu)$ as follows: $\Delta = \{b^q \mid b \in \Sigma, q \in Q\}$, $\mu(b^q) = b$ for every $b \in \Sigma$, and d consists of the rules $d(b^q) = r_{b,q}$ where $r_{b,q}$ is the regular expression obtained from $\delta(b, q)$ by replacing every occurrence of a state p by $(a^p + b^p)$. As every $t \in L(d)$ induces an accepting run of A on $\mu(t)$, it is immediate that A and D are equivalent.

From D, we now construct an EDTD D' such that

(i) if $L(A) = \mathcal{T}_\Sigma$ then $L(D')$ is defined by a DTD; and,
(ii) if $L(A) \neq \mathcal{T}_\Sigma$ then $L(D')$ is not defined by a restrained competition EDTD.

Of course (i) and (ii) together imply the statement of the theorem.

In fact D' is the union of the EDTDs D_1 and D_2 over the alphabet $\Gamma = \{a, b, \alpha, \beta, \text{root}\}$ defined next: D_1 accepts all trees of the form

$$\text{root}(\sigma(t_1 \cdots t_n))$$

where σ is α or β, $t_1, \ldots, t_n \in \mathcal{T}_\Sigma$, and the tree obtained from t_n by deleting its right-most leaf is accepted by A. Further, D_2 accepts all trees of the same form as for D_1, provided that, the right-most leaf of t_n is a (respectively, b) when σ is α (respectively, β). Note that D_1 can easily be constructed from D: D_1 just simulates D on the subtree rooted at the right most child of σ. The EDTD D_2 just needs to pass the symbol σ down to the right most leaf. Finally, define D' as the EDTD accepting $L(D_1) \cup L(D_2)$. Let $T = L(D')$.

We show (i) and (ii):

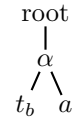
(i) First note that when $L(A) = \mathcal{T}_\Sigma$, then $L(D_2) \subseteq L(D_1)$ and T equals

$$\{\text{root}(\sigma(t_1 \cdots t_n)) \mid \sigma \in \{\alpha, \beta\}, t_1, \ldots, t_n \in \mathcal{T}_\Sigma\}.$$

The latter can clearly be defined by a DTD.

(ii) Let $L(A) \neq \mathcal{T}_\Sigma$ and let t be a tree not in $L(A)$. Towards a contradiction, assume that T is definable by a restrained competition EDTD. Let t_a and t_b be the trees obtained from t by adding an a and b respectively, to the right of the right-most leaf. Then $t'_a := \text{root}(\alpha(t_a)) \in T$ while $t'_b := \text{root}(\alpha(t_b)) \notin T$. Let t''_b be the tree obtained from t'_b by adding an a-leaf as right-most child of α, i.e.

$t''_b :=$



By definition of D', $t''_b \in T$. Let u be the right-most leaf of t'_a and let v be its parent. Then note that anc-sib-str$^{t'_a}(v) =$ anc-sib-str$^{t''_b}(v)$. So, by Theorem 9.1, $t'_a[v \leftarrow \text{subtree}^{t''_b}(v)]$ is in T when T is defined by a restrained competition EDTD. As the rightmost leaf of subtree$^{t''_b}(v)$ is a b, this implies that $t \in L(A)$, which is a contradiction. Hence, (ii) follows.

The exponential time upper bounds for the single-type and restrained competition cases can be obtained by performing the constructions in the proofs (c) ⇒ (a) and (d) ⇒ (b) in Theorems 7.1 and 9.1, respectively. Both the construction of the EDTD and checking equivalence with the original one can be done in exponential time. For DTDs a similar construction is in polynomial time but the equivalence check still needs exponential time.

—In the case of single-type EDTDs we proceed as follows. Let $D = (\Sigma, \Delta_D, d, s_d, \mu_D)$ be a given EDTD. We assume D is trimmed. We first construct the EDTD $E = (\Sigma, \Delta_E, e, s_d, \mu_E)$ as described in the proof of Theorem 7.1 (c) ⇒ (a). We argue that this can be done in EXPTIME. First, we need to compute $\Delta_E \subseteq 2^{\Delta_D}$. To this end, we enumerate all sets types(w). Let $s_d = c^0$. Initially, set $W := \{c\}$, Types(c) := $\{c^0\}$ and $R := \{\{c^0\}\}$.
Repeat the following until W becomes empty:
(1) Remove a string wa from W.
(2) For every $b \in \Sigma$, let Types(wab) contain all b^i for which there exists an a^j in Types(wa) and a string in $d(a^j)$ containing b^i. If Types(wab) is not empty and not already in R, then add it to R and add wab to W.

Since we add every set only once to R, the algorithm runs in time exponential in the size of D. Moreover, we have that Types(w) = types(w) for every w, and that $R = \Delta_E$. Now we know Δ_E, the rules of e can be directly computed.
It follows from the proof of Theorem 7.1 (c) ⇒ (a) that D is equivalent to a single-type EDTD iff D is in fact equivalent to E. Further, E then is the corresponding single-type EDTD. The construction of E can be done in exponential time and E might be of exponential size in D. Then it has to be checked whether D and E are equivalent. Fortunately, as always $L(D) \subseteq L(E)$, we only have to check

whether $L(E) - L(D)$ is empty. This involves the complementation of the tree automaton for D, resulting in a tree automaton of possibly exponential size, and in the test whether the automata for $L(E)$ and the complement of $L(D)$ have a non-empty intersection. The latter is polynomial in the size of the automata. Hence, we altogether get an exponential time algorithm.

—Testing whether an EDTD has an equivalent restrained competition EDTD can be done along the same lines, this time based on the proof of Theorem 9.1 (d) ⇒ (b). To compute types(w) for ancestor-sibling-strings w, we just need to let b in step (2) above range over $\Sigma \cup (\{\#\} \cdot \Sigma)$. A type b^ℓ is then added to Types(wb) if w is of the form $w'a\#x_1 \cdots x_k b$ where $x_1 \cdots x_k$ does not contain a separator # and

(1) there is an a^i in Types($w'a$) and
(2) there are $x^{i_j} \in$ Types($w'a\#x_1 \cdots x_j$),
(3) such that, $x_1^{i_1} \cdots x_k^{i_k} b^\ell$ is a prefix of a string in $d(a^i)$.

—Finally, we describe how it can be tested whether a given EDTD $D = (\Sigma, \Delta, d, s_d, \mu)$ has an equivalent DTD. As usual, we can assume that D is trimmed. Let, for each $a^i \in \Delta$, $r_{a,i}$ be the regular expression obtained from $d(a^i)$ by replacing every symbol b^j by b. We define a DTD (Σ, d_1, s_d) simply by taking the rules $a \to \bigcup_i r_{a,i}$, for every $a \in \Sigma$. It remains to show that D has an equivalent DTD if and only if $L(D) = L(d_1)$.

Analogously as in Theorem 7.1((c)⇒(a)), we have that $L(D) \subseteq L(d_1)$. Towards a contradiction, suppose that D has an equivalent DTD and that $t \in L(d_1) - L(D)$. According to Lemma 2.10 in [Papakonstantinou and Vianu 2000] (cf. Section 3.2), $L(D)$ is closed under label-guarded subtree exchange. As $t \notin L(D)$ there exists a node u in t such that subtree$^t(u) \notin L((D, a^i))$ for any $a^i \in \Delta$, but for every child $u_1, \ldots, u_n$ of u, we have that subtree$^t(u_j) \in L((D, b_j^{i_j}))$ for some $b_j^{i_j} \in \Delta$. Note that u and u_j are labeled with a and b, respectively. First, we note that u can never be a leaf node. Indeed, if there is no $a^i \in \Delta$ such that $\varepsilon \in L(r_{a,i})$, then ε is also not in $\bigcup_i L(r_{a,i})$, which is the content model of a in d_1.

If u is not a leaf node, we can do the following. By definition of d_1, for every $b_j^{i_j}$, there exists an a^k such that $b_j^{i_j}$ occurs in $d(a^k)$. Thus, as D is trimmed, for every u_j there exists a tree $t_j \in L(D)$ with a $v \in$ Nodes(t) such that lab$^{t_j}(v) = b_j$, the parent of v is labeled a, and subtree$^{t_j}(v) =$ subtree$^t(u)$. But this means that t can be constructed from $t_1, \ldots, t_n$ by label-guarded subtree exchange, which is a contradiction as $t \notin L(D)$.

■

10.3 Inclusion and Equivalence of Schemas

Decision problems like testing for inclusion or equivalence of schema languages often occur in schema optimization or as basic building blocks of algorithms for typechecking or type inference [Hosoya and Pierce 2003; Martens and Neven 2005; 2004; Papakonstantinou and Vianu 2000; Suciu 2001]. In general, these problems

are PSPACE and EXPTIME-complete for DTDs and EDTDs, respectively [Stockmeyer and Meyer 1973; Seidl 1990]. The XML specification, however, restricts regular expressions in DTDs to be deterministic [Bray et al. 2004] (sometimes also called 1-unambiguous [Brüggemann-Klein and Wood 1998], cf. Section 3.1).

THEOREM 10.4. *Given two restrained competition EDTDs D_1 and D_2, deciding whether*

(a) $L(D_1) \subseteq L(D_2)$*, and whether*
(b) $L(D_1) = L(D_2)$

is PSPACE*-complete in general, and* PTIME*-complete if D_1 and D_2 use deterministic regular expressions.*

Proof. Brüggemann-Klein and Wood formalized the notion of deterministic regular expressions and showed that a regular expression is deterministic iff its corresponding Glushkov automaton is a DFA [Brüggemann-Klein and Wood 1998]. We refrain from defining the Glushkov automaton corresponding to a regular expression but instead refer the interested reader to [Brüggemann-Klein and Wood 1998]. It suffices to know that for a given regular expression its Glushkov automaton can be computed in PTIME.

In the general case, the lower bounds are easy reductions from the inclusion and equivalence problems of regular expressions. Actually, it already holds for EDTDs which have only one non-trivial (not of the form $r \to \varepsilon$) and in which each element has only one type. Thus, the lower bounds also hold for single-type EDTDs (and even DTDs, if the requirement of being one-unambiguous is dropped). For deterministic expressions, the lower bound holds already for the non-emptiness problem and also even applies for DTDs without the requirement of being one-unambiguous (cf., e.g. [Stockmeyer and Meyer 1973; Martens et al. 2004; Martens and Neven 2005]).

For the upper bounds, it follows from Theorem 9.1 that for a tree language T defined by a restrained competition EDTD it holds that a tree t is in T if and only if $P_{\text{anc-sib}}(t)$ is in $P_{\text{anc-sib}}(T) := \{P_{\text{anc-sib}}(s) \mid s \in T\}$. Hence, $L(D_1) \subseteq L(D_2)$ if and only if $P_{\text{anc-sib}}(L(D_1)) \subseteq P_{\text{anc-sib}}(L(D_2))$.

For the upper bounds, (b) follows from (a), hence we only show (a). Given a restrained competition EDTD $D = (\Sigma, \Delta, d, s_d, \mu)$, we construct in polynomial time an NFA A for $P_{\text{anc-sib}}(L(D))$. For deterministic regular expressions A is a DFA. Testing inclusion of NFAs (DFAs) is well-known to be in PSPACE (PTIME).

Let for each $a^i \in \Delta$, $A_{a,i} = (Q_{a,i}, \Delta, \delta_{a,i}, s_{a,i}, F_{a_i})$ be an NFA (DFA) that defines $d(a^i)$ and has a unique state $q^\perp$ from which no final state is reachable. We can assume that each $A_{a,i}$ is *trimmed* in the sense that a final state is reachable from every state apart from $q^\perp$.

From the restrained competition property it immediately follows that in $A_{a,i}$, for each state q, if $\delta(q, b^j) = q_1$, $\delta(q, b^k) = q_2$, $q_1 \neq q_2$ and $j \neq k$ then q_1 or q_2 must be $q^\perp$. We require that the sets $Q_{a,i}$ are pairwise disjoint.

From these automata over the type set Δ we construct an automaton $A = (Q_A, \Sigma, s_A, \delta_A, F_A)$ as follows. The set Q_A consists of all pairs (q, b), where $q \in Q_{a,i}$, for some a^i, and $b \in \Delta \cup \{\#\}$. Intuitively, q is the current state of an automaton $A_{a,i}$ and b is the last extended symbol or type that has been identi-

fied. The initial state s_A of A is $(s_{b,j}, \#)$ for the initial symbol b^j of d. The transition function δ_A is defined as follows. For each $q \in Q_{a,i}$, $c \in \Delta \cup \{\#\}$ and $b \in \Sigma$ we let $\delta_A((q,c),b) = \{(p,b^j) \mid p \in \delta_{a,i}(q,b^j)\}$, for the unique j with $\delta_{a,i}(q,b^j) \neq q^\perp$, if such a j exists. Otherwise, $\delta_A((q,c),b) = (q^\perp, \#)$. Furthermore, we let $\delta_A((q,b^j),\#) = \{(s_{b,j},\#)\}$. We set $F_A = \{(q,c) \mid q \in F_{a,i}$ for some a, i and $c \in \Delta\}$. Note that A is a DFA if every $A_{a,i}$ is a DFA.

By construction, A accepts $P_{\text{anc-sib}}(L(D))$. It is easy to see that the size of A is no larger than the sum of the sizes of all $A'_{a,i}$. This concludes the proof. ∎

This result strongly contrasts with our results in [Martens et al. 2004], where we show that even for very simple non-deterministic regular expressions these decision problems are intractable, and with the case of arbitrary EDTDs with deterministic regular expressions, for which inclusion and equivalence test are EXPTIME-complete.

We end this section by a brief discussion on minimization of EDTDs. In general, their minimization is at PSPACE-hard and there is no unique minimal grammar. However, it is shown in [Martens and Niehren 2005] that for restrained competition and single-type EDTDs where regular languages are defined by DFAs, minimization is in PTIME. It then follows from a more general result on top-down tree automata that the resulting grammar is in fact unique up to isomorphism. Minimization of top-down deterministic unranked tree automata is also addressed in [Cristau et al. 2005].

11. SUBTREE-BASED SCHEMAS

From what was presented so far an obvious question arises. What happens if we soften the requirement that the type of an element has to be determined when its *opening* tag is visited? What if instead it has to be computed when the *closing* tag is seen? It turns out that every regular tree language has an EDTD which allows such 1-pass *postorder* typing. Furthermore, the EDTDs used for this purpose can be defined as straightforward extensions of restrained competition EDTDs.

Definition 11.1. An EDTD $D = (\Sigma, \Delta, d, s_d, \mu)$ is **extended restrained competition** iff for every regular expression r occurring in a rule the following holds: whenever there are two strings $w\tau v$ and $w\tau' v'$ in $L(r)$ with $\tau \neq \tau'$ and $\mu(\tau) = \mu(\tau')$, then $L((D,\tau)) \cap L((D,\tau'))$ is empty.

For a tree t and a node v, the **preceding-subtree** of v in t is the tree resulting from t by removing all right siblings of v and its ancestors together with the respective subtrees (cf. Figure 6). We denote the preceding-subtree of v by preceding-subtree$^t(v)$.

Definition 11.2. We say that an EDTD $D = (\Sigma, \Sigma', d, \mu)$ **has preceding-subtree-based types** if there is a function f which maps tree-node pairs to Σ' such that, for each tree $t \in L(D)$,

—t has exactly one witness t', and

—t' results from t by assigning to each node v the type $f(\text{preceding-subtree}^t(v), v)$.

Stated in terms of XML documents, the type of an element depends on the prefix of the document which ends with the closing tag of the element.

The following result shows that all regular tree languages admit 1-pass postorder typing.

THEOREM 11.3. *For a homogeneous tree language T the following are equivalent:*

(a) T is definable by an extended restrained competition EDTD;

(b) T is definable by an EDTD with preceding-subtree-based types; and,

(c) T is regular.

Proof. The directions (a) $\Rightarrow$ (c) and (b) $\Rightarrow$ (c) are trivial. The proof of the opposite directions uses the fact that regular languages can be validated by deterministic bottom-up automata.

(c) $\Rightarrow$ (a) and (c) $\Rightarrow$ (b): Let T be the tree language defined by a bottom-up deterministic tree automaton $B = (Q, \Sigma, \delta, F)$. We can assume that transition functions are represented by regular expressions. We construct an EDTD $D = (\Sigma, \Delta, d, s_d, \mu)$ such that $L(D) = L(B)$ exactly as in the proof of Theorem 10.3. In particular, $\Delta = \{a^q \mid a \in \Sigma, q \in Q\}$. It is immediate that a tree $t \in L(D, a^q)$ iff $\delta^*(t) = q$, where $\text{lab}^t(v) = a$ for the root v of t. Here, δ^* is the canonical extension of δ to trees. As B is deterministic, $L((D, a^q)) \cap L((D, a^{q'})) = \emptyset$ for all $a \in \Sigma$ and $q \neq q' \in Q$. Hence, D is extended restrained competition. By observing that there is only one accepting run for every tree and defining $f(\text{preceding-subtree}^t(u), u) = \delta^*(\text{subtree}^t(u))$, it follows that D has preceding-subtree-based types. ∎

In the EDTD used in the proof the type of each element actually only depends on its subtree. This should be compared with the previous characterizations where the type depended on the upper context.

Remark 11.4. Although there is an extended restrained competition for every regular tree language, not every EDTD itself is extended restrained competition. The EDTD D defined by the rules

$$r \to (a^1 + a^2) \qquad a^1 \to b + c + \varepsilon \qquad a^2 \to c + d + \varepsilon,$$

is *not* extended restrained competition, as $\{\varepsilon, c\} \subseteq L((D, a^1)) \cap L((D, a^2))$.

We conclude by noting that extended restrained competition is a tractable notion.

THEOREM 11.5. *It is decidable in* PTIME *for an EDTD D whether it is extended restrained competition.*

Proof. Let $D = (\Sigma, \Sigma', d, s_d, \mu)$ be an EDTD. Let E be the set $\{(a^i, a^j) \mid L((D, a^i)) \cap L((D, a^j)) \neq \emptyset\}$. This set can be computed in polynomial time by checking whether the non-deterministic tree automata for $L((D, a^i))$ and $L((D, a^j))$ have a non-empty intersection [Martens and Neven 2005].

It suffices to show that the following is in PTIME: testing whether for a single regular expression r there are two strings $w\tau v$ and $w\tau' v'$ in $L(r)$ with $\tau \neq \tau'$, $\mu(\tau) = \mu(\tau')$ and $L((D, \tau)) \cap L((D, \tau'))$ is empty. Let $N_r = (\Delta, Q, \delta, q_0, F)$ be an NFA equivalent to r.

The algorithm makes use of two sets:

—the set of reachable states $R := \{q \in Q \mid \exists w \in \Delta^*, \delta^*(q, w) \in F\}$; and,

—the set of pairs of states that can be reached by the same string, $S := \{(q_1, q_2) \in Q \times Q \mid \exists w \in \Delta^*, \{q_1, q_2\} \subseteq \delta^*(q_0, w)\}$.

Note that R and S can be computed in linear and quadratic time, respectively, by the usual reachability algorithm. Then, r is extended restrained competition iff there are no $q_1, q_2 \in S$ and a, i, j with $i \neq j$, $\delta(q_1, a^i) \cap R \neq \emptyset$, $\delta(q_2, a^j) \cap R \neq \emptyset$, and $(a^i, a^j) \in E$. The latter test is in PTIME. ∎

12. DISCUSSION

In this section, we present some concluding remarks. We start by making some concrete recommendations which directly follow from our results.

We have shown in Section 4, that the extra expressiveness of XML Schema over DTDs is only used to a very limited extent. A possible explanation is that users are simply not aware of what kind of context dependencies can be expressed within XML Schema. Our characterization in terms of ancestor-based schemas (Section 6.4), makes this ability explicit. To facilitate the use of these vertical patterns, we propose to add them as a conservative extension to XML Schema or develop a simple front-end based on DTDs as explained in Section 5.3 for less experienced XML users who might be discouraged by the high complexity of XML Schema.

We have argued that EDC does not capture the complete class of all efficiently typeable schemas. We have formalized the latter class as the EDTDs admitting 1PPT. Interestingly, the latter semantically defined class can be captured by EDTDs with restrained competition regular expressions. So the global constraint of 1PPT is characterized by a local constraint on regular expressions. Although restrained competition regular expressions are not syntactical, just like the one-unambiguous regular expressions characterizing UPA, a quadratic algorithm exists to recognize them. Just like for EDTDs with EDC, we provide a clear syntactical characterization in terms of ancestor-sibling-based schemas. This characterization makes explicit which context dependencies can be expressed while adhering to the 1PPT constraint. Again, these patterns can be added to XML Schema or can be incorporated in a front-end. So, for these reasons we propose to replace the EDC and the UPA constraints by restrained competition EDTDs.

In Section 8.7, we argued that both EDC and UPA already imply 1PPT (and therefore efficient typing). Thus, w.r.t. efficient typing, when adhering to UPA, it does not make much sense to also enforce EDC and vice-versa. It should be noticed that the class of EDTDs satisfying both EDC and UPA (like XML Schema) are a strict subclass of the EDTDs satisfying only one of EDC and UPA.

Although we think the restriction to unambiguous typing increases transparency and efficiency of validation, the recommendations in the present paper do not justify the former. For instance, Relax NG as well as the formal model for XML Schema of Siméon and Wadler [Siméon and Wadler 2003] allow ambiguous typing to relieve users from opaque restrictions and reaches the robust class of unranked regular tree languages which are closed under all Boolean operations. Especially in the context of data exchange it is of extreme importance that a schema language is closed under union (which is not the case for XML Schema). However, if unambiguous typing

and efficient processing is required, it should not be enforced by ad-hoc restrictions, but by the most liberal ones. We believe the restriction to 1-pass preorder typeable schemas is adequate. Moreover, it can be reached by allowing restrained competition regular expressions or by making use of the equivalent syntactic framework of ancestor-sibling-based schemas.

We already mentioned that Murata, Lee, and Mani already showed that DTD $\not\subseteq$ EDTD$^{\text{st}}$ $\not\subseteq$ EDTD$^{\text{rc}}$ $\not\subseteq$ EDTD [Murata et al. 2005]. They exhibited concrete tree languages that are in one class but not in the other. Our semantical characterizations provide tools to show inexpressibility for arbitrary tree languages. For instance, using the closure of restrained-competition EDTDs under ancestor-guarded subtree exchange, it is immediate that EDTD$^{\text{rc}}$ cannot define the set of all Boolean tree-shaped circuits evaluating to true.

Acknowledgments

We thank Nicole Schweikardt, Luc Segoufin, Dan Suciu, Jan Van den Bussche, and Stijn Vansummeren for helpful discussions. We thank the anonymous referees whose thorough comments and suggestions improved the presentation of the paper.

REFERENCES

Alur, R. and Madhusudan, P. 2004. Visibly pushdown languages. In *Proceedings of the 36th Symposium on the Theory of Computing (STOC)*. ACM Press, New York, 202–211.

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

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

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

Brüggemann-Klein, A., Murata, M., and Wood, D. 2001. Regular tree and regular hedge languages over unranked alphabets: Version 1, april 3, 2001. Tech. Rep. HKUST-TCSC-2001-0, The Hongkong University of Science and Technology.

Brüggemann-Klein, A. and Wood, D. 1998. One-unambiguous regular languages. *Information and Computation 142,* 2, 182–206.

Buck, L., Goldfarb, C., and Prescod, P. 2000. Datatypes for DTDs (DT4DTD) 1.0. Tech. rep., World Wide Web Consortium. January. http://www.w3.org/TR/dt4dtd/.

Clark, J. 2002. Multi-format schema converter based on RELAX NG. `http://www.thaiopensource.com/relaxng/trang.html`.

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

Coen, C. S., Marinelli, P., and Vitali, F. 2004. Schemapath, a minimal extension to XML Schema for conditional constraints. In *Proceedings of the 14th International Conference on World Wide Web (WWW)*. ACM Press, New York, 164–174.

Cover, R. 2005. The Cover pages. http://xml.coverpages.org/.

Cristau, J., Löding, C., and Thomas, W. 2005. Deterministic automata on unranked trees. To appear in FCT 2005.

DuCharme, B. 2002. Filling in the dtd gaps with schematron. O'Reilly xml.com.

Fernandez, M., Malhotra, A., Marsh, J., Nagy, M., and Walsh, N. 2005. XQuery 1.0 and XPath 2.0 data model. Tech. rep., World Wide Web Consortium. April. http://www.w3.org/TR/xpath-datamodel/.

Fiorello, D., Gessa, N., Marinelli, P., and Vitali, F. 2004. DTD++ 2.0: adding support for co-constraints. In *Extreme Markup Languages 2004*.

Fokoué, A. and Schloss, B. 2004. XML Schema quality checker. http://www.alphaworks.ibm.com/tech/xmlsqc.

Hopcroft, J. and Ullman, J. 1979. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley.

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

Hromkovic, J., Seibert, S., and Wilke, T. 2001. Translating regular expressions into small ε-free nondeterministic finite automata. *Journal of Computer and System Sciences 62,* 4, 565–588.

Jelliffe, R. 2001. The current state of the art of schema languages for XML. Presentation at XML Asia Pacific, Sidney, Australia.

Jelliffe, R. 2005. Schematron. http://xml.ascc.net/schematron/.

Klarlund, N., Moller, A., and Schwartzbach, M. I. 2000. The DSD schema language. In *Proceedings of the 3th ACM SIGSOFT Workshop on Formal Methods in Software Practice (FMSP)*.

Koch, C. and Scherzinger, S. 2003. Attribute grammars for scalable query processing on XML streams. In *Proceedings of the 9th International Workshop on Database Programming Languages (DBPL)*. Springer, Berlin, 233–256.

Lee, D. and Chu, W. 2000. Comparative analysis of six XML schema languages. *ACM SIGMOD Record 29,* 3, 76–87.

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

Martens, W. and Neven, F. 2004. Frontiers of tractability for typechecking simple XML transformations. In *Proceedings of the 23d Symposium on Principles of Database Systems (PODS)*. ACM Press, New York, 23–34.

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

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

Martens, W., Neven, F., and Schwentick, T. 2005. Which XML schemas admit 1-pass preorder typing? In *Proceedings of the 10th International Conference on Database Theory (ICDT)*. Springer, Berlin, 68–82.

Martens, W. and Niehren, J. 2005. Minimizing tree automata for unranked trees. In *Prodeedings of the 10th International Symposium on Database Programming Languages (DBPL 2005)*. 232–246.

Murata, M., Lee, D., and Mani, M. 2001. Taxonomy of XML schema languages using formal language theory. In *Extreme Markup Languages*. Montreal, Canada.

Murata, M., Lee, D., Mani, M., and Kawaguchi, K. 2005. Taxonomy of xml schema languages using formal language theory. *ACM Transactions on Internet Technology (TOIT) 5,* 4. To Appear.

Neven, F. 2002a. Automata, logic, and XML. In *Conference for Computer Science Logic (CSL)*. Springer, Berlin, 2–26.

Neven, F. 2002b. Automata theory for XML researchers. *SIGMOD Record 31,* 3, 39–46.

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

Papakonstantinou, Y. and Vianu, V. 2003. Incremental validation of XML documents. In *Proceedings of the 9th International Conference on Database Theory (ICDT)*. Springer, Berlin, 47–63.

Sahuguet, A. 2000. Everything you ever wanted to know about DTDs, but were afraid to ask. In *International Workshop on the Web and Databases (WebDB)*.

Segoufin, L. and Vianu, V. 2002. Validating streaming XML documents. In *Proceedings of the 21st Symposium on Principles of Database Systems (PODS)*. ACM Press, New York, 53–64.

Seidl, H. 1990. Deciding equivalence of finite tree automata. *SIAM Journal on Computing 19,* 3, 424–437.

Siméon, J. and Wadler, P. 2003. The essence of XML. In *30th Symposium on Principles of Programming Languages (POPL)*. ACM Press, New York, 1–13.

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

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

Stockmeyer, L. and Meyer, A. 1973. Word problems requiring exponential time: Preliminary report. In *Conference Record of Fifth Annual ACM Symposium on Theory of Computing (STOC)*. ACM Press, New York, 1–9.

Suciu, D. 2001. Typechecking for semistructured data. In *Proceedings of the 8th Workshop on Data Bases and Programming Languages (DBPL)*. Springer, Berlin, 1–20.

Thompson, H., Beech, D., Maloney, M., and Mendelsohn, N. 2004. XML Schema Part 1: Structures. Tech. rep., World Wide Web Consortium. October. http://www.w3.org/TR/xmlschema-1/.

van der Vlist, E. 2002. *XML Schema*. O'Reilly.

Vitali, F., Amorosi, N., and Gessa, N. 2003. Datatype- and namespace-aware DTDs: a minimal extension. In *Extreme Markup Languages*.

...