Simple off the shelf abstractions for XML Schema [*]

Wim Martens
University of Dortmund

Frank Neven
Hasselt University and
transnational University of Limburg

Thomas Schwentick
University of Dortmund

1 Introduction

Although the advent of XML Schema [25] has rendered DTDs obsolete, research on practical XML optimization is mostly biased towards DTDs and tends to largely ignore XSDs (some notable exceptions non-withstanding). One of the underlying reasons is most probably the perceived simplicity of DTDs versus the alleged impenetrability of XML Schema. Indeed, optimization w.r.t. DTDs has a local flavor and usually reduces to reasoning about the accustomed formalism of regular expressions. XSDs, on the other hand, even when sufficiently stripped down, are related to the less pervious class of unranked regular tree automata [6, 19, 20, 21]. Recent results on the structural expressiveness of XSDs [19], however, show that XSDs are in fact much closer to DTDs than to tree automata, leveraging the possibility to directly extend techniques for DTD-based XML optimization to the realm of XML Schema. The goal of the present paper is to present the results in [19] in an easy and accessible way. At the same time, we discuss possible applications, related research, and future research directions. Throughout the paper, we try to restrict notation to a minimum. We refer to [19] for further details.

2 DTDs versus XSDs

We informally discuss the difference in expressiveness between DTDs and XSDs. We borrow notation and some examples from [3]. For our purpose, an *XML fragment* is a (possibly empty) sequence `<`a_1`>` f_1 `</`a_1`>` $\ldots$ `<`a_n`>`f_n`</`a_n`>` of elements where $a_1, \ldots, a_n$ are *element names*, and $f_1, \ldots, f_n$ are themselves XML fragments. In particular, we ignore attributes and data values as we disregard schema features that constrain them.

[*]Database Principles Column. Column editor: Leonid Libkin.

Consider the XML document in Figure 1 that contains information about store orders and stock contents. Orders hold customer information and list the items ordered, with each item stating its id and price. The stock contents consists of the list of items in stock, with each item stating its id, the quantity in stock, and — depending on whether the item is atomic or composed from other items — some supplier information for the items of which they are composed, respectively. It is important to emphasize that order items do not include supplier information, nor do they mention other items. Moreover, stock items do not mention prices.

DTDs are incapable of distinguishing between order items and stock items because the content model of an element can only depend on the element's name in a DTD, and not on the context in which it is used. For example, although the DTD in Figure 2 describes all intended XML documents, it also allows supplier information to occur in order items and price information to occur in stock items.

The W3C specification [25] essentially defines an XSD as a collection of *type definitions*, which, if we abstract away from the concrete XML representation of XSDs, are rules like

$$\mathit{store} \rightarrow \texttt{order}[\mathit{order}]^*, \texttt{stock}[\mathit{stock}] \qquad (\star)$$

that map type names to regular expressions over pairs $a[t]$ of element names a and type names t. Throughout the article we use the convention that element names are typeset in `typewriter` font, and type names are typeset in *italic*. Intuitively, this particular type definition specifies an XML fragment to be of type *store* if it is of the form

$$\texttt{<order>}\, f_1 \,\texttt{</order>} \ldots \texttt{<order>}\, f_n \texttt{</order>} \quad \texttt{<stock>}\, g \,\texttt{</stock>}$$

where $n \geqslant 0$; $f_1, \ldots, f_n$ are XML fragments of type *order*; and g is an XML fragment of type *stock*. Each type name that occurs on the right hand side of a type definition in an XSD must also be defined in the XSD, and each type name may be defined only once.

Using types, an XSD can specify that an item is an order item when it occurs under an order element and is a stock item otherwise. For example, Figure 3 shows an XSD describing the intended set of store documents. Notice in particular the use of the types $item_1$ and $item_2$ to distinguish between order items and stock items.

It is important to remark that the 'Element Declaration Consistent' (EDC) constraint of the W3C specification requires multiple occurrences of the same element name in a single type definition to occur with the same type. Hence, type definition ($\star$) is legal, but

$$persons \to (\texttt{person}[male] + \texttt{person}[female])^+$$

is not, as `person` occurs both with type *male* and type *female*. Of course, element names in *different* type definitions can occur with different types (which is exactly what yields the ability to let the content model of an element depend on its context).

3 A formal abstraction

Fix a finite set EName and Types of element and type names, respectively. The set of *elements* is then defined as $\mathsf{Elem}(\mathsf{EName}, \mathsf{Types}) = \{a[t] \mid a \in \mathsf{EName}, t \in \mathsf{Types}\}$. As EName and Types will be always clear from the context, we simply write Elem in the sequel.

We view an XML fragment $f = f_1 \cdots f_n$ as a sequence of labeled trees where every tree consists of a finite number of nodes, and every node v is assigned an element name denoted by $\mathrm{lab}(v)$. We assume the

```
<store>
  <order>
    <customer>
       <name>John Mitchell</name>
       <email> j.mitchell@yahoo.com </email>
    </customer>
    <item> <id> I18F </id>
           <price> 100 </price>
    </item>
    <item> ... </item> ... <item> ... </item>
  </order>
  <order> ... </order> ... <order> ... </order>
  <stock>
    <item>
      <id> IG8 </id> <qty> 10 </qty>
      <supplier> <name> Al Jones </name>
                 <email> a.j@gmail.com </email>
                 <email> a.j@dot.com </email>
       </supplier>
     </item>
     <item>
       <id> J38H </id> <qty> 30 </qty>
       <item>
          <id> J38H1 </id> <qty> 10 </qty>
          <supplier> ... </supplier>
       </item>
       <item>
          <id> J38H2 </id> <qty> 1 </qty>
          <supplier> ... </supplier>
      </item>
      <item> ... </item> ... <item> ... </item>
   </item>
   ...
   <item> ... </item>
 </stock>
</store>
```

Figure 1: Example XML document.

```
<!ELEMENT store (order*, stock)>
<!ELEMENT order (customer, item+)>
<!ELEMENT customer (name, email*)>
<!ELEMENT item (id, (price +
              (qty, (supplier + item+))))>
<!ELEMENT stock (item+)>
<!ELEMENT supplier (name, email*)>
```

Figure 2: A DTD describing the document in Figure 1.

$$\begin{array}{lll}
\textit{root} & \rightarrow & \texttt{store}[\textit{store}] \\
\textit{store} & \rightarrow & \texttt{order}[\textit{order}]^*, \texttt{stock}[\textit{stock}] \\
\textit{order} & \rightarrow & \texttt{customer}[\textit{person}], \texttt{item}[\textit{item}_1]^+ \\
\textit{person} & \rightarrow & \texttt{name}[\textit{emp}], \texttt{email}[\textit{emp}]^+ \\
\textit{item}_1 & \rightarrow & \texttt{id}[\textit{emp}], \texttt{price}[\textit{emp}] \\
\textit{stock} & \rightarrow & \texttt{item}[\textit{item}_2]^+ \\
\textit{item}_2 & \rightarrow & \texttt{id}[\textit{emp}], \texttt{qty}[\textit{emp}], \\
 & & \quad (\texttt{supplier}[\textit{person}] + \texttt{item}[\textit{item}_2]^+) \\
\textit{emp} & \rightarrow & \varepsilon
\end{array}$$

Figure 3: An XSD describing the XML document in Figure 1. The symbol ε denotes the empty string.

existence of a virtual root root which acts as the common parent of the roots of the different f_i.

The set of regular expressions r, denoted by REG, is given by the following syntax:

$$r ::= \varepsilon \mid \alpha \mid r, r \mid r + r \mid r^* \mid r^+ \mid r?$$

where ε denotes the empty string and $\alpha \in \mathsf{Elem}$. Their semantics is the usual one and is therefore omitted.[1]

Definition 1. An *XSchema* is a tuple $S = (\mathsf{EName}, \mathsf{Types}, \rho, t_0)$ where EName and Types are finite sets of elements and types, respectively, ρ is a mapping from Types to regular expressions over alphabet Elem, and, $t_0 \in \mathsf{Types}$ is the start type.

We sometimes also refer to $\rho(t)$ as the *content model associated to t*. Later on, we are going to restrict ρ to *deterministic* regular expressions as defined below in Section 4.

Example 1. In Figure 3, $\mathsf{EName} = \{\texttt{store}, \texttt{order}, \texttt{stock}, \texttt{customer}, \texttt{item}, \texttt{name}, \texttt{email}, \texttt{id}, \texttt{qty}, \texttt{price}, \texttt{supplier}\}$, $\mathsf{Types} = \{\textit{root}, \textit{store}, \textit{order}, \textit{person}, \textit{item}_1, \textit{item}_2, \textit{stock}, \textit{emp}\}$, $t_0 = \textit{root}$, and the function ρ is depicted in arrow notation.

A *typing* τ of f is a mapping assigning a type $\tau(v) \in \mathsf{Types}$ to every node v in f (including the virtual root). For a node v with children $v_1, \ldots, v_n$, define child-string(τ, v) as the typed string $\mathrm{lab}(v_1)[\tau(v_1)] \cdots \mathrm{lab}(v_n)[\tau(v_n)]$.

Definition 2 (validation)**.** An XML fragment f *conforms to* or is *valid w.r.t.* a schema $S = (\mathsf{EName}, \mathsf{Types}, \rho, t_0)$, if there is a typing τ of f such that, for every node v, child-string(τ, v) matches the regular expression $\rho(\tau(v))$, and $\tau(\mathsf{root}) = t_0$. We then call τ a *valid typing.*

Despite the clean formalization, the above definition does not entail a validation algorithm. One possibility is to compute, for each node v in f, a set of possible types $\Delta(v) \subseteq \mathsf{Types}$ such that, for each type $t \in \Delta(v)$, the XML subfragment rooted at v is valid w.r.t. the schema with start type t. The XML fragment is then valid w.r.t. S itself when the start type t_0 belongs to $\Delta(\mathsf{root})$. The sets $\Delta(v)$ can be computed in a bottom-up fashion. Indeed, $t \in \Delta(v)$ iff (1) v is a leaf node and $\rho(t)$ contains the empty string; or, (2) v is a non-leaf node with children $v_1, \ldots, v_n$ and there are $t_1 \in \Delta(v_1), \ldots, t_n \in \Delta(v_n)$ such that $\mathrm{lab}(v_1)[t_1] \cdots \mathrm{lab}(v_n)[t_n] \in \rho(t)$. A valid typing can then be computed from the sets Δ by an additional top-down pass through the tree. Although this kind of bottom-up validation is a bit at odds with the general concept of top-down or streaming XML processing, the algorithm can be adapted to this end (cf., for instance, [20, 22]).

Before we restrict XSchemas to obtain the corresponding classes of DTDs and XSDs, we first discuss deterministic regular expressions.

4 Deterministic regular expressions

Not only the occurrence of types in rules is restricted by the XML Schema specification, but also the shape of the regular expressions in the rules themselves. That is, regular expressions should be deterministic. This constraint is often referred to as UPA: the Unique Particle Attribution constraint. The intuition behind the constraint is the following: the form of the regular expression should allow to match each symbol of the input string uniquely against a position in the expression when processing the input

[1]We note that XSDs actually allow numerical occurrence operators (`minoccurs` and `maxoccurs`) and a mild form of shuffling (`ALL`). As these are all definable within REG, we disregard them for the moment.

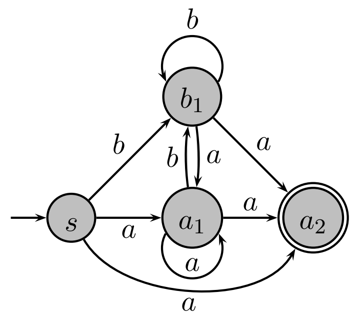
string in one pass from left to right. That is, without looking ahead in the string. For instance, the expression $r_1 = (a+b)^*a$ is not deterministic as already the first symbol in the string aaa can be matched to two different a's in r_1. The equivalent expression $r_2 = b^*a(b^*a)^*$, on the other hand, is deterministic. Unfortunately, not every non-deterministic regular expression can be rewritten into an equivalent deterministic one [5]. Thus, semantically, the class of deterministic regular expressions, which we denote here by DREG, is a strict subclass of the regular languages. Moreover, it is not very robust, as it is not closed under union, concatenation, or Kleene-star, prohibiting an elegant constructive definition [5].

There has been quite some debate in the XML community about the restriction to deterministic regular expressions (cf., e.g., pg 98 of [26] and [17, 24]) as it does not serve its purpose: even for general regular expressions simple validation algorithms exist that are as efficient as those for deterministic regular expressions. One reason to maintain this restriction is to ensure compatibility with SGML parsers, the predecessor of XML.

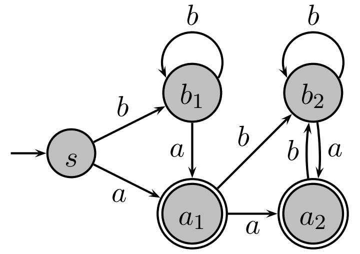
Deterministic regular expressions are characterized as *one-unambiguous* regular expressions by Brüggemann-Klein and Wood [5]. For a regular expression r over elements, we denote by $\overline{r}$ the regular expression obtained from r by replacing, for each i, the ith a-element in r (counting from left to right) by a_i. For example, $\overline{r_2}$ is $b_1^*a_1(b_2^*a_2)^*$.

Definition 3. A regular expression r is *one-unambiguous* iff there are no strings wa_iv and wa_jv' in $L(\overline{r})$ so that $i \neq j$.

Deciding whether a regular expression r is one-unambiguous can be done in quadratic time [4]. The algorithm constructs the *Glushkov Automaton* $G(r)$ for r and checks whether it is deterministic. In a nutshell, the states of $G(r)$ are the positions of $\overline{r}$ plus an initial state s. There is a transition from position x_i to y_j if there is a string in which the successive symbols x, y can be matched to x_i and y_j, respectively. A state is accepting if the corresponding position can match the final symbol of a word. The Glushhov automata $G(r_1)$ and $G(r_2)$ are depicted in Figure 4.



(a)



(b)

Figure 4: The Glushkov automata $G(r_1)$ and $G(r_2)$. Note that $G(r_2)$ is deterministic whereas $G(r_1)$ is not.

It can be decided in EXPTIME whether there is a deterministic regular expression equivalent to a given regular expression [5]. If so, the algorithm can return an equivalent deterministic expression of a size which is double exponential.

5 DTDs and XSDs formalized

We restrict the general class of XSchemas to DTDs and XSDs:

Definition 4. Let $S = (\mathsf{EName}, \mathsf{Types}, \rho, t_0)$ be a schema. Then,

1. S is *local* when $\mathsf{EName} = \mathsf{Types}$ and regular expressions in ρ are defined over the alphabet $\{a[a] \mid a \in \mathsf{EName}\}$; this simply means that the name of the element also functions as its type.

2. S is *single-type* when there are no elements $a[t_1]$ and $a[t_2]$ in a $\rho(t)$ with $t_1 \neq t_2$.

We now formally define the different classes of XSchemas (as proposed in [20]):

Definition 5.
- A *DTD* is a local XSchema with deterministic regular expressions.
- An *XSD* is a single-type XSchema with deterministic regular expressions.

A Relax NG schema [7] can then be abstracted by an XSchema.

6 Typing a schema

For general XSchemas, a valid typing is not necessarily unique. Consider for instance the schema

$$\begin{array}{lcl} \textit{root} & \rightarrow & \mathsf{a}[\textit{a1}] + \mathsf{a}[\textit{a2}] \\ \textit{a1} & \rightarrow & \mathsf{b}[\textit{emp}] \\ \textit{a2} & \rightarrow & \mathsf{b}[\textit{emp}] \\ \textit{emp} & \rightarrow & \varepsilon \end{array}$$

defining the fragment `<a><b/></a>` where a can be both assigned the type *a1* and *a2*. In addition, computing a valid typing can not be achieved in one top-down pass through the XML fragment. Consider for instance the schema

$$\begin{array}{lcl} \textit{root} & \rightarrow & \mathsf{a}[\textit{a1}] + \mathsf{a}[\textit{a2}] \\ \textit{a1} & \rightarrow & \mathsf{b}[\textit{emp}] \\ \textit{a2} & \rightarrow & \mathsf{c}[\textit{emp}] \\ \textit{emp} & \rightarrow & \varepsilon \end{array}$$

No type can be assigned to a before its child is visited. In contrast, the single-type restriction ensures that XSDs can be uniquely typed in a top-down fashion. To be precise, one-pass typing in a top-down fashion means that the first time a node is visited a type should be assigned (so only based on what has been seen up to now) and that a child can be visited only when its parent is already visited.

Theorem 1. *[20, 19] When an XML fragment f is valid w.r.t. an XSD, then there is exactly one valid typing which in addition can be computed in a one-pass top-down fashion.*

Proof. The theorem follow from a simple algorithm to validate an XML fragment against a schema $S = (\mathsf{EName}, \mathsf{Types}, \rho, t_0)$. Define $\tau(\mathsf{root}) = t_0$. For every node v with children $v_1, \ldots, v_n$ for which $\tau(v)$ is defined, let t_i be the unique type such that $\mathrm{lab}(v_i)[t_i]$ occurs in $\tau(v)$. Set $\tau(v_i) = t_i$. When $\text{child-string}_\tau(v) \notin \rho(\tau(v))$ then reject as the document is not valid, otherwise proceed as before. $\square$

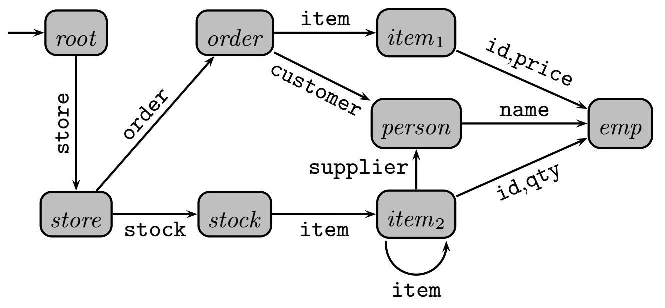
Theorem 1 has an interesting consequence. In a scenario where XML data is processed as a stream, the type of each element is determined when its opening tag arrives. Consequently, any decisions depending on the type of an element can be triggered immediately. Similarly, parsing w.r.t. an XSD works fine for documents in SAX-representation.

We mention that when UPA is enforced, the single-type or EDC constraint is actually not necessary to obtain unique one-pass top-down typing: UPA alone already implies it (cf. Section 8.7 in [19]). The reason being that any deterministic regular expression is *restrained-competition* and the latter implies one-pass preorder and therefore also top-down typing. Actually, the class of restrained-competition XSchemas captures precisely the fragment of XSchemas admitting one-pass preorder typing [19].

7 A type-concealed definition of XSDs: DFA-based XSDs

From the proof of Theorem 1, it already becomes apparent that the type of a node in a valid typing w.r.t. an XSD $S = (\mathsf{EName}, \mathsf{Types}, \rho, t_0)$ only depends on the type of its parent. That type in turn only depends on its parent, and so on until the root is reached. Actually, this type dependence can be captured by a deterministic finite automaton (DFA). Indeed, define a DFA which starts at the root in initial state/type t_0 and moves from state/type t to state/type t' while reading a iff $a[t']$ occurs in $\rho(t)$. This view decouples types from the rules and hides them in the automaton. We formalize this next. In this respect, let child-string(v) be the string formed by the labels of the children of v.

Definition 6. A *DFA-based* XSD is a tuple $D = (\mathsf{EName}, A, \lambda)$, where A is a DFA using the states



$$\begin{aligned}
\lambda(root) &= \texttt{store}\\
\lambda(store) &= \texttt{order}^*\texttt{stock}\\
\lambda(order) &= \texttt{customer item}^+\\
\lambda(person) &= \texttt{name email}^+\\
\lambda(item_1) &= \texttt{id price}\\
\lambda(stock) &= \texttt{item}^+\\
\lambda(item_2) &= \texttt{id qty (supplier} + \texttt{item}^+)\\
\lambda(emp) &= \varepsilon
\end{aligned}$$

Figure 5: A DFA-based XSD equivalent to the XSD in Figure 3.

Types and λ is a function mapping states of A to regular expressions in DREG over EName (so not over Elem!). An XML fragment f is *valid w.r.t. D* if, for every node v of f, child-string$(v) \in \lambda(q)$, where q is the state reached by A when started in its start state on the path from root to v.

A DFA-based XSD for our running example is displayed in Figure 5.

The following Proposition (which is proved as Lemma 7 in [11]) shows that the model of DFA-based XSDs can be used without compromise in modeling XML Schema.

Theorem 2. Any DFA-based XSD can be translated into an equivalent XSD in at most quadratic time, and vice versa.

8 A type-free definition of XSDs: pattern-based XSDs

As DTDs do not employ types, the content model of a node is determined by its label. So, the context which can be delineated by a DTD is simply the element name of the node at hand. For a node v, we denote by anc-str(v) the *ancestor-string* which is given by the labels of the nodes on the path from root to v. From the discussion in the previous section, it becomes apparent that the context which can be described by an XSD is restricted to the ancestor-string of the node at hand and can be defined in a regular way. By replacing the DFA in Definition 6 by regular expressions, we obtain a formalism closely related to DTDs [19].

Definition 7. A *pattern-based XSD P* is a set $\{r_1 \hookrightarrow s_1, \ldots, r_m \hookrightarrow s_m\}$ of rules, where all r_i are in REG and all s_i are in DREG.

We refer to the r_i and s_i as the vertical and the horizontal patterns, respectively. The following two semantics for pattern-based XSDs have been considered [13].

Definition 8.
- An XML fragment f is *existentially valid* with respect to a pattern-based schema P if, for every node v of f, there is a rule $r \hookrightarrow s \in P$ such that anc-str(v) matches r and child-string(v) matches s.
- An XML fragment f is *universally valid* with respect to a pattern-based schema P if, for every node v of f, and each rule $r \hookrightarrow s \in P$ it holds that anc-str(v) matches r implies child-string(v) matches s.

A pattern-based schema for our running example is shown in Figure 6. The reader might notice that in this example the existential and the universal semantics coincide. Though more convenient as a specification mechanism than DFA-based XSDs, translation to and from XSDs is a bit more problematic as shown by the following Theorem. In Section 9 we exhibit fragments occurring in practice with better behavior.

Theorem 3.
1. Translating a pattern-based XSD under the existential or universal semantics to an equivalent XSD requires double exponential time [11].
2. Translating an XSD to an equivalent pattern-based XSD under the existential or universal semantics requires exponential time [19].

$$
\begin{array}{rcl}
\varepsilon & \looparrowright & \mathtt{store} \\
\mathtt{store} & \looparrowright & \mathtt{order}^{*}\ \mathtt{stock} \\
\mathtt{store\ order} & \looparrowright & \mathtt{customer}\ \mathtt{item}^{+} \\
\mathtt{store\ order\ customer} & \looparrowright & \mathtt{name}\ \mathtt{email}^{+} \\
\mathtt{store\ order\ item} & \looparrowright & \mathtt{id\ price} \\
\mathtt{store\ stock} & \looparrowright & \mathtt{item}^{+} \\
\mathtt{store\ stock\ item}^{+} & \looparrowright & \mathtt{id\ qty}\ (\mathtt{supplier} + \mathtt{item}^{+}) \\
\mathtt{store\ stock\ item}^{+}\ \mathtt{supplier} & \looparrowright & \mathtt{name}\ \mathtt{email}^{+}
\end{array}
$$

$$
\begin{array}{rcl}
\mathtt{store\ order\ item}\ (\mathtt{id}+\mathtt{price}) & \looparrowright & \varepsilon \\
\mathtt{store\ order\ customer}\ (\mathtt{name}+\mathtt{email}) & \looparrowright & \varepsilon \\
\mathtt{store\ stock\ item}^{+}\ \mathtt{supplier}\ (\mathtt{name}+\mathtt{email}) & \looparrowright & \varepsilon \\
\mathtt{store\ stock\ item}^{+}\ (\mathtt{id}+\mathtt{qty}) & \looparrowright & \varepsilon
\end{array}
$$

Figure 6: A pattern-based XSD for the store example.

9 XSDs in practice

The formal taxonomy presented in Definition 5 begs the question to what extent the expressiveness of DTDs and XSDs is actually used in practice. In [19, 2], a substantial corpus of DTDs and XSDs was harvested from the Web, including the Cover Pages [9] incorporating high-quality schemas representing various standards such as the XML Schema Specification, XHTML, UDDI, RDF and others. The study in [19] mainly focused on expressiveness in terms of typing while [2] together with [1] also considered content models.

9.1 Local Typing

It turns out that out that the far majority (85%) of the considered XSDs where in fact structurally equivalent to a DTD: at most one type is associated to every element name. So only the remaining 15% of the XSDs use the typing mechanism to actually define non-local classes of XML documents. Surprisingly, in 90% of these cases, types only depend on the parent context like in Figure 3 where an `item` has type $item_1$ when its parent has label `order` and type $item_2$ otherwise. In the few remaining cases, types depend on the grand- or the great grand-parent context as for instance exemplified in Figure 7. The interpretation 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.

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

Figure 7: An XSD abstracted from the most complicated XSD found in [19]: the type of j-elements depends on their great grand-parent.

9.2 Content models

In [1] it was noted that in most regular expressions each element name occurs at most once. This observation lead to the definition of *single occurrence regular expressions (SOREs).* For instance, $a((b^* + c)^* d)^*$ is a SORE while $a(a + b)^*$ is not as a occurs twice. An earlier look at the same corpus of DTDs and XSDs in [2] revealed that most (99%) regular expressions occurring in practical schemas are in fact *chain regular expressions (CHAREs).*[2] Each such expression is a SORE which can be written as a sequence of factors $f_1 \cdots f_n$ where every factor is an expression of the form $(a_1 + \cdots + a_k)$, $(a_1 + \cdots + a_k)?$, $(a_1 + \cdots + a_k)^+$, or $(a_1 + \cdots + a_k)^*$. Here, $k \geqslant 1$ and every a_i is an element name. For instance, the expression $a(b + c)^* d^+ (e + f)?$ is a CHARE, while $(ab + c)^*$ and $(a^* + b?)^*$ are not. Note that every SORE, and therefore also every CHARE is deterministic (or one-unambiguous) as required by the XML specification.

9.3 Implications

The discussion above implies that a large portion of practical XSDs is captured by the fragment of pattern-based XSDs where all vertical patterns are restricted to $//w$ and all horizontal patterns are SOREs. Here, $//$ is XPath's descendant axis and w is a path of element names. The pattern-based XSD of Figure 6 using this notation is depicted in Figure 8.

In [3] algorithms for learning this practical subclass of XSDs have been proposed. Furthermore, in

[2]The single-occurrence property was initially missed.

```
              ε ↳ store
         //store ↳ order* stock
         //order ↳ customer item+
      //customer ↳ name email+
    //order/item ↳ id price
         //stock ↳ item+
    //stock/item ↳ id qty(supplier + item+)
     //item/item ↳ id qty (supplier + item+)
      //supplier ↳ name email+

   //id ↳ ε          //qty ↳ ε        //price ↳ ε
 //name ↳ ε        //email ↳ ε
```

Figure 8: A pattern-based XSD for the store in XPath notation.

strong contrast to general pattern-based schemas (cf. Theorem 3), when assuming a mild disjointness criterion, translating between existential and universal semantics, and translating back and forth to single-type XSchemas can be done in polynomial time [11].

10 Inexpressibility

Let t_1, t_2 be two valid XML fragments for a DTD d and let v_1 and v_2 be nodes of t_1 and t_2, respectively, with the same element name a. It is not hard to see that the fragment resulting from replacing the subtree t'_1 rooted at v_1 in t_1 by the subtree t'_2 rooted at v_2 in t_2 is again valid w.r.t. d. We say that DTDs (or the sets of fragments they define) have the *label-guarded subtree exchange property.* Figure 9 gives an illustration.

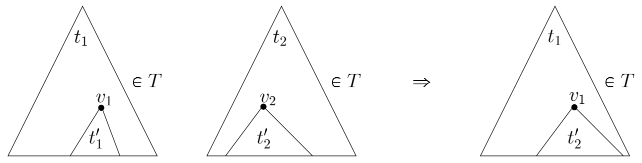


Figure 9: Label-guarded subtree exchange. Nodes v_1 and v_2 are both labeled with the same label.

It turns out that XSDs also have a subtree exchange property, but this time it is *ancestor-guarded*, i.e., a subtree exchange can take place if v_1 and v_2 have the same ancestor-string.

The importance of the characterization of XSDs by a subtree-exchange property stems from the fact that inexpressibility results can be formally proved rather than vaguely stated: a set of XML fragments lacking this property can not be characterized by any XSD. For instance, a shortcoming attributed to XSDs is their inability to express certain co-constraints [8]. A simple example of such a co-constraint is the following: there must be an `order`-element with at least two `item`-children. Using the ancestor-guarded subtree exchange property, it is very easy to prove that this co-constraint cannot be expressed with XSDs. Indeed, let f_1 and f_2 be two XML fragments with two orders each. In f_1 the first order has two items and the second order has one item. In f_2 the first order has one items and the second order has two items. By replacing the first order of f_1 by the first order of f_2 we obtain an XML fragment without two-item orders.

Finally, in the same spirit it can be easily shown that XSDs, just as DTDs, lack some of the basic closure properties: they are not closed under union nor under negation.

11 Optimization

Because of the correspondence with regular tree automata, the inclusion and equivalence of XSchemas is EXPTIME-complete [23], even when regular expressions are restricted to be deterministic [18]. For single-type XSchemas, these decision problems reduce to the corresponding decision problems on the class of allowed regular expressions [18, 19] and are therefore in polynomial time for XSDs. Furthermore, given an XSchema, it can be decided in EXPTIME whether an equivalent XSD or DTD exist. If so, an equivalent schema can also be constructed in EXPTIME [19].

12 Conclusions

We presented a detailed account of the structural expressiveness of XSDs. The most important message being that, in contrast to what is mostly assumed, XML Schema is much closer to DTDs than to tree automata. In brief, it can be seen as DTDs extended with vertical regular expressions. Furthermore, both vertical and horizontal expressions can be greatly simplified to capture all practical XSDs.

An important omission from the abstraction presented here are the counting and shuffling expressions allowed in content models. These have a serious impact on the complexity of decision problems [10, 12, 14]. Moreover, one-unambiguity for such expressions is not yet fully understood [15, 16].

Acknowledgments

We thank Wouter Gelade for his comments.

References

[1] G.J. Bex, F. Neven, T. Schwentick, and K. Tuyls. Inference of concise DTDs from XML data. In *VLDB*, pages 115–126, 2006.

[2] G.J. Bex, F. Neven, and J. Van den Bussche. DTDs versus XML Schema: A practical study. In *WebDB*, pages 79–84, 2004.

[3] G.J. Bex, F. Neven, and S. Vansummeren. Inferring XML Schema Definitions from XML data. In *VLDB*, pages 998–1009, 2007.

[4] A. Brüggemann-Klein. Regular expressions into finite automata. *Theoretical Computer Science*, 120(2):197–213, 1993.

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

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

[7] J. Clark and M. Murata. *RELAX NG Specification.* OASIS, December 2001.

[8] C. Sacerdoti Coen, P. Marinelli, and F. Vitali. Schemapath, a minimal extension to XML Schema for conditional constraints. In *WWW*, pages 164–174, 2004.

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

[10] W. Gelade, W. Martens, and F. Neven. Optimizing schema languages for XML: Numerical constraints and interleaving. In *ICDT*, pages 269–283, 2007.

[11] W. Gelade and F. Neven. Succinctness of pattern-based schema languages for XML. In *DBPL*, 2007.

[12] G. Ghelli, D. Colazzo, and C. Sartiani. Efficient inclusion for a class of XML types with interleaving and counting. In *DBPL*, 2007.

[13] G. Kasneci and T. Schwentick. The complexity of reasoning about pattern-based XML schemas. In *PODS*, pages 155–164, 2007.

[14] P. Kilpeläinen and R. Tuhkanen. Regular expressions with numerical occurrence indicators – preliminary results. In *SPLST*, pages 163–173, 2003.

[15] P. Kilpeläinen and R. Tuhkanen. Towards efficient implementation of XML schema content models. In *DOCENG*, pages 239–241, 2004.

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

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

[18] W. Martens, F. Neven, and T. Schwentick. Complexity of decision problems for simple regular expressions. In *MFCS*, pages 889–900, 2004.

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

[20] M. Murata, D. Lee, M. Mani, and K. Kawaguchi. Taxonomy of XML schema languages using formal language theory. *ACM Transactions on Internet Technology*, 5(4):1–45, 2005.

[21] F. Neven. Automata theory for XML researchers. *SIGMOD Record*, 31(3):39–46, 2002.

[22] L. Segoufin and V. Vianu. Validating streaming XML documents. In *PODS*, pages 53–64, 2002.

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

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

[25] C.M. Sperberg-McQueen and H. Thompson. XML Schema. Technical report, World Wide Web Consortium, 2005. http://www.w3.org/XML/Schema.

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