http://webdam.inria.fr/

Web Data Management

Typing Semistructured Data

Serge Abiteboul
INRIA Saclay & ENS Cachan

Ioana Manolescu
INRIA Saclay & Paris-Sud University

Philippe Rigaux
CNAM Paris & INRIA Saclay

Marie-Christine Rousset
Grenoble University

Pierre Senellart
Télécom ParisTech

Copyright @2011 by Serge Abiteboul, Ioana Manolescu, Philippe Rigaux, Marie-Christine Rousset, Pierre Senellart;
to be published by Cambridge University Press 2011. For personal use only, not for distribution.

http://webdam.inria.fr/Jorge/

Contents

1 Motivating Typing **2**

2 Automata **5**
2.1 Automata on Words 5
2.2 Automata on Ranked Trees 6
2.3 Unranked Trees 8
2.4 Trees and Monadic Second-Order Logic 10

3 Schema Languages for XML **10**
3.1 Document Type Definitions 10
3.2 XML Schema 13
3.3 Other Schema Languages for XML 16

4 Typing Graph Data **18**
4.1 Graph Semistructured Data 18
4.2 Graph Bisimulation 18
4.3 Data guides 19

5 Further reading **19**

6 Exercises **21**

In this chapter, we discuss the typing of semistructured data. Typing is the process of describing, with a set of declarative rules or constraints called a *schema*, a class of XML documents, and verifying that a given document is *valid* for that class (we also say that this document is valid *against* the type defined by the schema). This is for instance used to define a specific XML vocabulary (XHTML, MathML, RDF, etc.), with its specificities in structure and content, that is used for a given application.

We first present motivations and discuss the kind of typing that is needed. XML data typing is typically based on finite-state automata. Therefore, we recall basic notion of automata, first on words, then on ranked trees, finally on unranked trees, i.e., essentially on XML. We also present the main two practical languages for describing XML types, DTDs and XML Schema, both of which endorsed by the W3C. We then briefly describe alternative schema languages with their key features. In a last section, we discuss the typing of graph data.

One can also consider the issue of “type checking a program”, that is, verifying that if the input is of a proper input type, the program produces an output that is of a proper output type. In Section 5, we provide references to works on program type checking in the context of XML.

1 Motivating Typing

Perhaps the main difference with typing in relational systems is that typing is not compulsory for XML. It is perfectly fine to have an XML document with no prescribed type. However, when developing and using software, types are essential, for interoperability, consistency, and efficiency. We describe these motivations next and conclude the section by contrasting two kinds of type checking, namely dynamic and static.

@ Serge Abiteboul, Ioana Manolescu, Philippe Rigaux, Marie-Christine Rousset, Pierre Senellart, 2011; to be published by Cambridge University Press 2011.

Interoperability. Schemas serve to document the interface of software components, and provide therefore a key ingredient for the interoperability between programs: a program that consumes an XML document of a given type can assume that the program that has generated it has produced a document of that type.

Consistency. Similarly to dependencies for the relational model (primary keys, foreign key constraints, etc.), typing an XML document is also useful to protect data against improper updates.

Storage Efficiency. Suppose that some XML document is very regular, say, it contains a list of companies, with, for each, an ID, a name, an address and the name of its CEO. This same information may be stored very compactly, for instance, without repeating the names of elements such as `address` for each company. Thus, *a priori* knowledge on the type of the data may help improve its storage.

Query Efficiency. Consider the following XQuery query:

```
for $b in doc("bib.xml")/bib//*
where $b/*/zip = '12345'
return $b/title
```

Knowing that the document consists of a list of `book`s and knowing the exact type of `book` elements, one may be able to rewrite the query:

```
for $b in doc("bib.xml")/bib/book
where $b/address/zip = '12345'
return $b/title
```

that is typically much cheaper to evaluate. Note that in the absence of a schema, a similar processing is possible by first computing from the document itself a *data guide*, i.e., a structural summary of all paths from the root in the document. There are also other more involved *schema inference* techniques that allow attaching such an *a posteriori* schema to a schemaless document.

Dynamic and Static Typing

Assume that XML documents (at least some of them) are associated with schemas and that programs use these schemas. In particular, they verify that processed documents are valid against them. Most of the time, such verification is dynamic. For instance, a Web server verifies the type when sending an XML document or when receiving it. Indeed, XML data tend to be checked quite often because programs prefer to verify types dynamically (when they transfer data) than risking to run into data of unexpected structure during execution.

It is also interesting, although more complicated, to perform static type checking, i.e., verify that a program receiving data of the proper input type only generates data of the proper output type. More formally, let note $d \models T$ when a document d is valid against a type T. We

@ Serge Abiteboul, Ioana Manolescu, Philippe Rigaux, Marie-Christine Rousset, Pierre Senellart, 2011; to be published by Cambridge University Press 2011.

say that a type T_1 is *more specific* than a type T_2 if all documents valid against T_1 are also valid against T_2, i.e.,

$$\forall d\,(d \models T_1 \Rightarrow d \models T_2).$$

Let T_i be an *input type* and f be a program or query that takes as input an XML document and returns an XML document. This f might be an XPath or XQuery query, an XSLT stylesheet, or even a program in a classical programming language. Static typing implies either static verification or static inference, defined as follows:

Verification: Is it true that $\forall d \models T_i$, $f(d) \models T_o$, for some given *output type* T_o?

Inference: Find the *most specific* T_o such that $\forall d \models T_i$, $f(d) \models T_o$.

Note that in a particular case, we have no knowledge of the input type, and the inference problem becomes: Find the *most specific* T_o such that $f(d) \models T_o$.

The notion of *smallest output type* depends of the schema language considered. Assume for instance that types are described as regular expressions on the tag sequences[1] and consider the following XQuery query:

```
for $p in doc("parts.xml")//part[color="red"]
return <part>{$p/name, $p/desc}</part>
```

Assuming no constraint on the input type, it is easy to see that the type of the result is described by the following regular expression:

$$(\texttt{<part>}\ (\texttt{<name>}\ \mathit{any}\ \texttt{</name>})^*\ (\texttt{<desc>}\ \mathit{any}\ \texttt{</desc>})^*\ \texttt{</part>})^*$$

where *any* stands for any well-formed tag sequence. The regular language described by this regular expression is also the smallest one that describes the output of the query, since any document in this language can be generated as the output of this query.

Verifying or inferring an output type for a program is in all generality an undecidable problem, even for XPath queries and a simple schema language such as DTDs. Even when the verification problem is decidable, a smallest output type might not exist. Consider for instance the XQuery program:

```
let $d:=doc("input.xml")
for $x in $d//a, $y in $d//a
return <b/>
```

Suppose that the input is `<input><a/><a/></input>`. Then the result is:

```
<b/><b/><b/><b/>
```

[1]As shall be seen later in this chapter, typical schema languages for XML are more adapted to the tree structure of the document because they are defined in terms of regular tree languages rather than of regular string languages.

@ Serge Abiteboul, Ioana Manolescu, Philippe Rigaux, Marie-Christine Rousset, Pierre Senellart, 2011; to be published by Cambridge University Press 2011.

In general, the result consists in n^2 `b`-elements for some $n \geq 0$. Such a type cannot be described by DTDs or XML schemas. One can approximate it by regular expressions but not obtain a "best" result:

$$
\begin{array}{l}
\texttt{<b/>}^* \\
\epsilon + \texttt{<b/>} + \texttt{<b/>}^4\texttt{<b/>}^* \\
\epsilon + \texttt{<b/>} + \texttt{<b/>}^4 + \texttt{<b/>}^9\texttt{<b/>}^* \\
\ldots
\end{array}
$$

2 Automata

XML was recently introduced. Fortunately, the model can benefit from a theory that is well established, automata theory. We briefly recall some standard definitions and results on automata over words. Then we mention without proof how they extend to ranked trees with some limitations. As previously mentioned, XML is based on unranked trees, so we finally consider unranked trees.

2.1 Automata on Words

This is standard material. We recall here briefly some notation and terminology.

Definition 2.1 *A* finite-state word automaton *(FSA for short) is a 5-tuple* $(\Sigma, Q, q_0, F, \delta)$ *where*

1. Σ *is a finite alphabet;*
2. Q *is a finite set of states;*
3. $q_0 \in Q$ *is the initial state;*
4. $F \subseteq Q$ *is the set of final states; and*
5. δ*, the transition function, is a mapping from* $(\Sigma \cup \{\epsilon\}) \times Q$ *to* 2^Q.

Such a nondeterministic automaton accepts or rejects words in Σ^*. A word $w \in \Sigma^*$ is accepted by an automaton if there is a path in the state space of the automaton, leading from the initial state to one of the final states and compatible with the transition functions, such that the concatenation of all labels from $\Sigma \cup \{\epsilon\}$ along the path is w. The set of words accepted by an automaton A is denoted $L(A)$. A language accepted by an FSA is called a *regular* language. They can alternatively be described as regular expressions built using concatenation, union and Kleene closure, e.g., $a(b+c)^*d$.

Example 2.2 Consider the FSA A with $\Sigma = \{a, b\}$, $Q = \{q_0, q_1, q_2, q_3\}$, $F = \{q_2\}$, $\delta(a, q_0) = \{q_0, q_1\}$, $\delta(b, q_1) = \{q_0\}$, $\delta(\epsilon, q_1) = \{q_2\}$, $\delta(\epsilon, q_2) = \{q_3\}$, $\delta(\epsilon, q_3) = \{q_3\}$. Then *abaab* is not in $L(A)$ whereas *aba* is in $L(A)$.

An automaton is *deterministic* if (i) it has no ϵ-transition, i.e., $\delta(\epsilon, q) = \emptyset$ for all q; and (ii) there are no multiple transitions from a single pair of symbol and state, i.e., $|\delta(a, q)| \leq 1$ for all (a, q).

The following important results are known about FSAs:

@ Serge Abiteboul, Ioana Manolescu, Philippe Rigaux, Marie-Christine Rousset, Pierre Senellart, 2011; to be published by Cambridge University Press 2011.

1. For each FSA A, one can construct an equivalent deterministic FSA B (i.e., a deterministic FSA accepting exactly the same words). In some cases, the number of states of B is necessarily exponential in that of A. This leads to another fundamental problem that we will ignore here, the problem of state minimization, i.e., the problem of finding an equivalent deterministic automaton with as few states as possible.

2. There is no FSA accepting the language $\{a^i b^i \mid i \geq 0\}$.

3. Regular languages are closed under complement.
(To see this, consider an automaton A. Construct a deterministic automaton that accepts the same language but never "blocks", i.e., that always reads the entire word. Let Q be the set of states of this automaton and F its set of accepting states. Then the automaton obtained by replacing F by $Q - F$ for accepting states, accepts the complement of the language accepted by A.)

4. Regular languages are closed under union and intersection.
(Given two automata A, B, construct an automaton with states $Q \times Q'$ that simulates both. For instance, an accepting state for $L(A) \cap L(B)$ is a state (q, q'), where q is accepting for A and q' for B.)

2.2 Automata on Ranked Trees

Automata on words are used to define word languages, that is, subsets of Σ^* for some alphabet Σ. Similarly, it is possible to define *tree automata* whose purpose is to define subsets of the set of all trees. For technical reasons, it is easier to define tree automata for ranked trees, i.e., trees whose number of children per node is bounded. We will explain in Section 2.3 how the definitions can be extended to unranked trees.

Word automata defined in the previous section process a string from left to right. It is easy to define a notion of right-to-left automaton, and also easy to see that in terms of accepted languages, there is absolutely no difference between left-to-right and right-to-left automata. For trees, there is a difference between top-down and bottom-up automata. Intuitively, in a top-down automaton, we have a choice of the direction to go (e.g., to choose to go to the first child or the second) whereas in bottom-up automata, similarly to word automata, the direction is always prescribed.

Bottom-Up Tree Automata. Let us start with the example of bottom-up automata for binary trees. Similarly to word automata, a bottom-up automaton on binary trees is defined by:

1. A finite *leaf* alphabet $\mathcal{L}$;

2. A finite *internal* alphabet Σ, with $\Sigma \cap \mathcal{L} = \emptyset$;

3. A set of states Q;

4. A set of accepting states $F \subseteq Q$;

5. A transition function δ that maps:

    - a leaf symbol $l \in \mathcal{L}$ to a set of states $\delta(l) \subseteq 2^Q$;

@ Serge Abiteboul, Ioana Manolescu, Philippe Rigaux, Marie-Christine Rousset, Pierre Senellart, 2011; to be published by Cambridge University Press 2011.

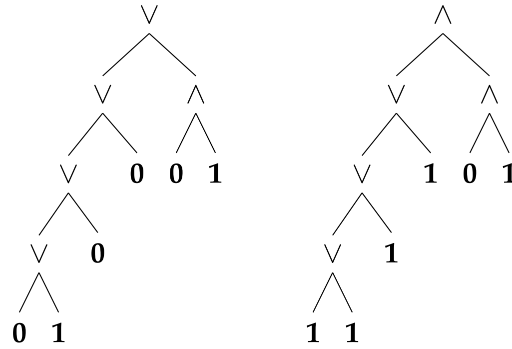

Figure 1: Two binary trees

- an internal symbol $a \in \Sigma$, together with a pair of states (q,q') to a set of states $\delta(a,q,q') \subseteq 2^Q$.

The transition function specifies a set of state for the leaf nodes. Then if $\delta(a,q,q')$ contains q'', this specifies that if the left and right children of a node labeled a are in states q,q', respectively, then the node *may move* to state q''.

Example 2.3 [Boolean circuit] Consider the following bottom-up automaton: $\mathcal{L} = \{0,1\}$, $\Sigma = \{\vee,\wedge\}$, $Q = \{f,t\}$, $F = \{t\}$, and $\delta(0) = \{f\}$, $\delta(1) = \{t\}$, $\delta(\wedge,t,t) = \{t\}$, $\delta(\wedge,f,t) = \delta(\wedge,t,f) = \delta(\wedge,f,f) = \{f\}$, $\delta(\vee,f,f) = \{f\}$, $\delta(\vee,f,t) = \delta(\vee,t,f) = \delta(\vee,t,t) = \{t\}$. The tree of the left of Figure 1 is accepted by the bottom-up tree automata, whereas the one on the right is rejected. More generally, this automaton accepts and/or trees that evaluate to true.

The definition can be extended to ranked trees with symbols of arbitrarily arity in a straightforward manner. An ϵ-transition in the context of bottom-up tree automata is of the form $\delta(a,r) = r'$, meaning that if a node of label a is in state r, then it may move to state r'. We can also define deterministic bottom-up tree automata by forbidding ϵ-transition and alternatives (i.e., some $\delta(a,q,q')$ containing more than one state).

Definition 2.4 *A set of trees is a* regular tree language *if it is accepted by a bottom-up tree automata.*

As for automata on words, one can "determinize" a nondeterministic automata. More precisely, given a bottom-up tree automata, one can construct a deterministic bottom-up tree automata that accepts the same trees.

Top-Down Tree Automata. In a top-down tree automaton, transitions are of the form $(q,q') \in \delta(a,q'')$ with the meaning that if a node labeled a is in state q'', then this transition moves its left child to state q and its right child to q'. The automaton accepts a tree if *all* leaves can be set in accepting states when the root is in some given initial state q_0. Determinism is defined in the obvious manner.

It is not difficult to show that a set of trees is regular if and only if it is accepted by a top-down automata. On the other hand, deterministic top-down automata are weaker.

@ Serge Abiteboul, Ioana Manolescu, Philippe Rigaux, Marie-Christine Rousset, Pierre Senellart, 2011; to be published by Cambridge University Press 2011.

Consider the language $L = \{f(a,b), f(b,a)\}$. It is easy to see it is regular. Now one can verify that if there is a deterministic top-down automata accepting it, then it would also accept $f(a,a)$, a contradiction. Thus deterministic top-down automata are weaker.

Generally speaking, one can show for regular tree languages the same results as for regular languages (sometimes the complexity is higher). In particular:

1. Given a tree automata, one can find an equivalent one (bottom-up only) that is deterministic (with possibly an exponential blow-up).

2. Regular tree languages are closed under complement, intersection and union (with similar proofs than for word automata).

2.3 Unranked Trees

We have defined in Section 2.2 tree automata (and regular tree languages) over the set of *ranked trees*, i.e., trees where there is an *a priori* bound of the number of children of each node. But XML documents are unranked (take for example XHTML, in which the number of paragraphs `<p>` inside the body of a document is unbounded).

Reconsider the Boolean circuit example from Example 2.3. Suppose we want to allow and/or gates with arbitrary many inputs. The set of transitions of a bottom-up automaton becomes infinite:

$$\begin{array}{lll} \delta(\wedge,t,t,t) = t, & \delta(\wedge,t,t,t,t) = t, & \ldots \\ \delta(\wedge,f,t,t) = f, & \delta(\wedge,t,f,t) = f, & \ldots \\ \delta(\vee,f,f,f) = f, & \delta(\vee,f,f,f,f) = f, & \ldots \\ \delta(\vee,t,f,f) = t, & \delta(\vee,f,t,f) = t, & \ldots \end{array}$$

So an issue is to represent this infinite set of transitions. To do that, we can use regular expressions on words.

Example 2.5 Consider the following regular word languages:

$$\begin{array}{ll} And_1 = tt^* & And_0 = (t+f)^* f (t+f)^* \\ Or_0 = ff^* & Or_1 = (t+f)^* t (t+f)^* \end{array}$$

Then one can define infinite sets of transitions:

$$\delta(\wedge, And_1) = \delta(\vee, Or_1) = t, \qquad \delta(\wedge, And_0) = \delta(\vee, Or_0) = f$$

One can base a theory of unranked trees on that principle. Alternatively, one can build on ranked trees by representing any unranked tree by a binary tree where the left child of a node represents the first child and the right child, its next sibling in the original tree, as shown in Figure 2.

Let F be the one-to-one mapping that encodes an unranked tree T into $F(T)$, the binary tree with first-child and next-sibling. Let F^{-1} be the inverse mapping that "decodes" binary trees thereby encoded. One can show that for each unranked tree automata A, there exists a ranked tree automata accepting $F(L(A))$. Conversely, for each ranked tree automata A, there is an unranked tree automata accepting $F^{-1}(L(A))$. Both constructions are easy.

As a consequence, one can see that unranked tree automata are closed under union, intersection and complement.

@ Serge Abiteboul, Ioana Manolescu, Philippe Rigaux, Marie-Christine Rousset, Pierre Senellart, 2011; to be published by Cambridge University Press 2011.

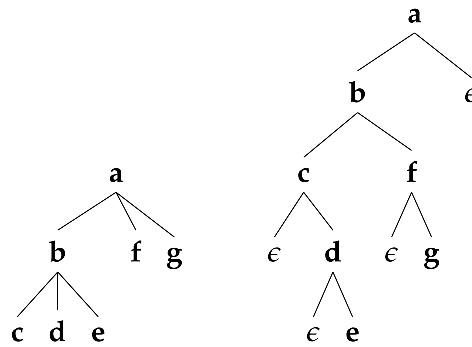

Figure 2: An unranked tree and its corresponding ranked one

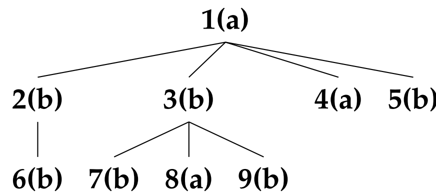

Figure 3: A tree with node identifiers listed

Determinism. Last, let us consider determinism for unranked tree automata. We cannot build on the translation to ranked tree automata as we did for union, intersection and complement. This is because the translation between ranked and unranked trees does not preserve determinism. So, instead we define determinism directly on bottom-up tree automata over unranked trees.

Let us define more precisely the machinery of these automata, and first the nondeterministic ones. The automata over unranked trees are defined as follows. An automata A includes a finite alphabet Σ of labels, a finite set Q of states, and a set F of accepting states. For each $a \in \Sigma$, it includes an automaton A_a over words that takes both its word alphabet and its states in Q. Consider a tree T with labels in Σ. Suppose its root is labelled a and that it has $n \geq 0$ subtrees, $T_1, ..., T_n$, in that order. Then A may reach state q on input T if there exists $q_1, ...q_n$ such that:

- For each i, A may reach the state q_i on input T_i.
- A_a may reach state q on input $q_1...q_n$.

The automaton A accepts T if q is accepting.

Now for the automaton A to be deterministic, we need to prevent the possibility that it may reach two states for the same input tree. For this, we require A_a to be deterministic for each a.

@ Serge Abiteboul, Ioana Manolescu, Philippe Rigaux, Marie-Christine Rousset, Pierre Senellart, 2011; to be published by Cambridge University Press 2011.

2.4 Trees and Monadic Second-Order Logic

There is also a logical interpretation of regular tree languages in terms of *monadic second-order logic*. One can represent a tree as a logical structure using identifiers for nodes. For instance the tree of Figure 3, where "1(a)" stands for "node id 1, label a", is represented by:

$$\begin{array}{l} E(1,2), E(1,3), \ldots, E(3,9) \\ S(2,3), S(3,4), S(4,5), \ldots, S(8,9) \\ a(1), a(4), a(8) \\ b(2), b(3), b(5), b(6), b(7), b(9) \end{array}$$

Here, the predicate $E(x,y)$ denotes the *child* relation, while the predicate $S(x,y)$ represents the *next-sibling* relation.

The syntax of *monadic second-order logic* (MSO) is given by:

$$\varphi \quad :- \quad x = y \mid E(x,y) \mid S(x,y) \mid a(x) \mid ... \mid \varphi \wedge \varphi \mid \varphi \vee \varphi \mid \neg\varphi \mid \exists x\, \varphi \mid \exists X\, \varphi \mid X(x)$$

where x is an atomic variable and X a variable denoting a set. $X(x)$ stands for $x \in X$. Observe that in $\exists X\, \varphi$, we are quantifying over a set, the main distinction with first-order logic where quantification is only on atomic variables. The term *monadic* means that the only second-order variables are unary relations, that is, sets.

MSO is a very general logical languages for trees. For instance, we can capture in MSO the constraint: "each a node has a b descendant". This is achieved by stating that for each node x labeled a, each set X containing x and closed under descendants contains some node labeled b. Formally,

$$\forall x\, a(x) \rightarrow (\forall X\, X(x) \wedge \beta(X) \rightarrow \exists y\, X(y) \wedge b(y))$$

where $\beta(X) = \forall y \forall z\, (X(y) \wedge E(y,z) \rightarrow X(z))$

We state a last result in this section because it beautifully illustrates the underlying theory. We strongly encourage the reader to read further about tree automata and about monadic second-order logic.

Theorem 2.6 A set L of trees is regular if and only if $L = \{T | T \models \varphi\}$ for some monadic second-order formula φ, i.e., if L is definable in MSO.

3 Schema Languages for XML

In this section, we present actual languages that are used to describe the type of XML documents. We start with DTDs, that are part of the specification of XML, and then move to the other schema language endorsed by the W3C, XML Schema. We finally discuss other existing schema languages, highlighting the differences with respect to DTDs and XML Schema.

3.1 Document Type Definitions

DTD stands for "Document Type Definition". An integral part of the XML specification, this is the oldest syntax for specifying typing constraints on XML, still very much in use. To

@ Serge Abiteboul, Ioana Manolescu, Philippe Rigaux, Marie-Christine Rousset, Pierre Senellart, 2011; to be published by Cambridge University Press 2011.

describe types for XML, the main idea of DTDs is to describe the children that nodes with a certain label may have. With DTDs, the labels of children of a node of a given label are described by regular expressions. The syntax, inherited from SGML, is bizarre but rather intuitive.

An example of a DTD is as follows:

```
<!ELEMENT populationdata (continent*) >
<!ELEMENT continent (name, country*) >
<!ELEMENT country (name, province*)>
<!ELEMENT province ((name|code), city*) >
<!ELEMENT city (name, pop) >
<!ELEMENT name (#PCDATA) >
<!ELEMENT code (#PCDATA) >
<!ELEMENT pop (#PCDATA) >
```

The comma is the concatenation operator, indicating a sequence of children, while "|" is the union operator that expresses alternatives. "*" indicates an arbitrary number of children with that structure; `#PCDATA` just means "textual content". In DTDs, the regular expressions are supposed to be *deterministic*. In brief, the XML data can be parsed (and its type verified) by a deterministic finite-state automaton that is directly derived from the regular expression. For instance, the expression

$$(a+b)^*a$$

(in DTD syntax, `((a|b)*,a)`) is not deterministic since when parsing a first a, one doesn't know whether this is the a of $(a+b)^*$ or that of the last a. On the other hand, this expression is equivalent to

$$(b^*a)^+$$

(in DTD syntax, `((b*,a)+)`), that is deterministic.

Under this restriction, it is easy to type-check some XML data while scanning it, e.g., with a SAX parser. Observe that such a parsing can be sometimes performed with a finite-state automaton but that sometimes more is required. For instance, consider the following DTD:

```
<!ELEMENT part (part*) >
```

The parser reads a list of *part* opening tags. A stack (or a counter) is needed to remember how many where found to verify that the same number of closing tags is found.

The reader may be a bit confused by now. Is this language regular (an automaton suffices to validate it) or not? To be precise, if we are given the tree, a tree automaton suffices to check whether a document satisfies the previous DTD. On the other hand, if we are given the serialized form of the XML document, just the verification that the document is well-formed cannot be achieved by an automaton. (It requires a stack as $a^n b^n$ does.)

Observe that we do want the kind of recursive definitions that cannot be verified simply by an FSA. On the other hand, DTDs present features that are less desired, most importantly, they are not closed under union:

@ Serge Abiteboul, Ioana Manolescu, Philippe Rigaux, Marie-Christine Rousset, Pierre Senellart, 2011; to be published by Cambridge University Press 2011.

DTD_1:
```
<!ELEMENT used (ad*) >
<!ELEMENT ad (year, brand) >
```

DTD_2:
```
<!ELEMENT new (ad*) >
<!ELEMENT ad (brand) >
```

$L(DTD_1) \cup L(DTD_2)$ cannot be described by a DTD although it can be described easily with a tree automaton. The issue here is that the type of *ad* depends of its parent. We can approximate what we want:

```
<!ELEMENT ad (year?, brand) >
```

But this is only an approximation. It turns out that DTDs are also not closed under complement.

What we need to do is to decouple the notions of type and that of label. Each type corresponds to a label, but not conversely. So, for instance, we may want two types for ads, with the same label:

```
car:  [car]  (used|new)*
used: [used] (ad1*)
new:  [new]  (ad2*)
ad1:  [ad]   (year, brand)
ad2:  [ad]   (brand)
```

With such decoupling, we can prove closure properties. This is leading to XML Schema, described in the next section, that is based on decoupled tags with many other features.

DTDs provide a few other functionalities, such as the description of the type of attributes, a mechanism for including external files and for defining local macros, or the possibility of declaring an attribute value as unique (`ID`) in the whole document or refers to one of these unique values (`IDREF`). These `ID` and `IDREF` attribute types can be used to create "pointers" from one point of the document to another, as investigated in Section 4. XML documents can include in-line description of their DTDs, as well as refer to external DTDs to indicate their type. Thus, XHTML documents typically contain such a declaration (before the opening tag of the root element and after the optional XML declaration):

```
<!DOCTYPE html
  PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
         "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
```

Here, `html` is the name of the root element, `-//W3C//DTD XHTML 1.0 Strict//EN` is a *public identifier* for this DTD (that Web browsers can for instance use to identify the particular HTML dialect a document is written in) and `http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd` is the *system identifier*, a URL to the DTD.

A final annoying limitation of DTDs, overcome in XML Schema, is their unawareness of XML namespaces. Because XML namespaces are very much used in practice, this considerably limits the usefulness of DTDs. In brief, it is impossible to give a DTD for an XML document that will remain valid when the namespace prefixes of this document are renamed. Let us explain this further. An XML document making use of namespaces is usually conceptually the

@ Serge Abiteboul, Ioana Manolescu, Philippe Rigaux, Marie-Christine Rousset, Pierre Senellart, 2011; to be published by Cambridge University Press 2011.

same when the prefix used to refer to the namespace is changed.[2] For instance, the following three documents are usually considered to be syntactic variants of the same:

```
<a xmlns="http://toto.com/"><b /></a>
<t:a xmlns:t="http://toto.com/"><t:b /></t:a>
<s:a xmlns:s="http://toto.com/"><s:b /></s:a>
```

These documents can be distinguished using advanced features (*namespace nodes*) of XML programming interfaces and languages (DOM, SAX, XPath, XSLT, XQuery, etc.) but it is very rarely done or useful. As DTDs have been introduced at the same time that XML itself, they predate the introduction of namespaces in XML. A side-effect of this is that it is impossible to write a DTD all three documents above are valid against. In XML Schema (as well as in other modern schema languages for XML), this is directly supported.

3.2 XML Schema

XML Schema is an XML-based language for describing XML types proposed by the W3C. Despite criticism for being unnecessarily complicated, this is the primary schema language for XML documents (disregarding DTDs), notably because of its support and use in other W3C standards (XPath 2.0, XSLT 2.0, XQuery 1.0, WSDL for describing Web services, etc.). In essence, XML schemas are very close to deterministic top-down tree automata but, as already mentioned, with many practical gadgets. It uses an XML syntax, so it benefits from XML tools such as editors and type checkers.

An example XML schema is given in Figure 4. XML schemas first include the definition of simple elements with atomic types, where the common types are `xs:string`, `xs:decimal`, `xs:integer`, `xs:boolean`, `xs:date`, `xs:time`. For instance, one can define:

```
<xs:element name="lastname" type="xs:string"/>
<xs:element name="age" type="xs:integer"/>
<xs:element name="dateborn" type="xs:date"/>
```

And corresponding data are:

```
<lastname>Refsnes</lastname>
<age>34</age>
<dateborn>1968-03-27</dateborn>
```

One can also define attributes as, for instance in:

```
<xs:attribute name="lang" type="xs:language"/>ă
```

with for corresponding data:

```
<lastname lang="en-US">Smith</lastname>
```

[2]There are exceptions to that, when the namespace prefix is also used in attribute values or text content, such as in XSLT or XML Schema, but they are rare.

@ Serge Abiteboul, Ioana Manolescu, Philippe Rigaux, Marie-Christine Rousset, Pierre Senellart, 2011; to be published by Cambridge University Press 2011.

```
<?xml version="1.0" encoding="utf-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="book">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="title" type="xs:string"/>
        <xs:element name="author" type="xs:string"/>
        <xs:element name="character"
                    minOccurs="0" maxOccurs="unbounded">
          <xs:complexType>
            <xs:sequence>
              <xs:element name="name" type="xs:string"/>
              <xs:element name="friend-of" type="xs:string"
                          minOccurs="0" maxOccurs="unbounded"/>
              <xs:element name="since" type="xs:date"/>
              <xs:element name="qualification" type="xs:string"/>
            </xs:sequence>
          </xs:complexType>
        </xs:element>
      </xs:sequence>
      <xs:attribute name="isbn" type="xs:string"/>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

Figure 4: Simple XML schema

@ Serge Abiteboul, Ioana Manolescu, Philippe Rigaux, Marie-Christine Rousset, Pierre Senellart, 2011; to be published by Cambridge University Press 2011.

One can impose restrictions of simple elements as in:

```
<xs:element name="age">
  <xs:simpleType>
    <xs:restriction base="xs:integer">
      <xs:minInclusive value="0"/>
      <xs:maxInclusive value="100"/>
    </xs:restriction>
  </xs:simpleType>
</xs:element>
```

Other restrictions are: enumerated types, patterns defined by regular expressions, etc.

XML schemas also allow defining *complex elements* that possibly correspond to subtrees of more that one nodes. A complex element may be empty, contain text, other elements or be "hybrid", i.e., contain both some text and subelements.

One can define the content of complex elements as in:

```
<xs:element name="person">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="firstname" type="xs:string"/>
      <xs:element name="lastname" type="xs:string"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

It is also possible to specify a type and give it a name.

```
<xs:complexType name="personinfo">
  <xs:sequence> <xs:element name="firstname" type="xs:string"/>
  <xs:element name="lastname" type="xs:string"/> </xs:sequence>
</xs:complexType>
```

Then we can use this type name in a type declaration, as in:

```
<xs:element name="employee" type="personinfo" />
```

One should also mention some other useful gadgets:

1. It is possible to import types associated to a namespace.

```
<xs:import nameSpace = "http://..."
   schemaLocation = "http://..." />
```

2. It is possible to include an existing schema.

```
<xs:include schemaLocation="http://..." />
```

@ Serge Abiteboul, Ioana Manolescu, Philippe Rigaux, Marie-Christine Rousset, Pierre Senellart, 2011; to be published by Cambridge University Press 2011.

```
<!ELEMENT book (title, author, character*) >
<!ELEMENT title (#PCDATA) >
<!ELEMENT author (#PCDATA) >
<!ELEMENT character (name,friend-of*,since,qualification)>
<!ELEMENT name (#PCDATA) >
<!ELEMENT name friend-of (#PCDATA) >
<!ELEMENT since (#PCDATA) >
<!ELEMENT qualification (#PCDATA) >

<!ATTLIST book isbn CDATA #IMPLIED >
```

Figure 5: Example DTD, corresponding to the XML schema of Figure 4

3. It is possible to extend or redefine an existing schema.

```
<xs:redefine schemaLocation="http://..." />
   ... Extensions ...
</xs:redefine>
```

There are more restrictions on XML schemas, some rather complex. For instance, inside an element, no two types may use the same tag. Some of them can be motivated by the requirement to have efficient type validation (i.e., that the top-down tree automaton defined by the schema is deterministic). The main difference with DTDs (besides some useful gadgets) is that the notions of types and tags are decoupled.

XML Schema also allows going beyond what is definable by tree automata with *dependencies*: it is possible to define *primary keys* (`<xs:key />`) and *foreign keys* (`<xs:keyref />`, in a similar way as keys on relational databases. This extends and complements the `xs:ID` and `xs:IDREF` datatypes that XML Schema inherits from DTDs.

To conclude and contrast DTDs with XML Schema, consider again the XML Schema from Figure 4. A corresponding DTD is given in Figure 5. In this example, the only difference between the DTD and the XML schema is that the DTD is unable to express the datatype constraint on `since`.

3.3 Other Schema Languages for XML

DTDs and XML Schema are just two examples of schema languages that can be used for typing XML documents, albeit important ones because of their wide use and their endorsement by the W3C. We briefly present next other approaches.

RELAX NG. RELAX NG (REgular LAnguage for XML Next Generation) is a schema language for XML, spearheaded by the OASIS consortium, and is a direct concurrent to XML Schema, with which it shares a number of features. The most striking difference is at the syntax level, since RELAX NG provides, in addition to an XML syntax, a non-XML syntax, which is much more compact and readable than that of XML Schema. As an example, here is a RELAX NG schema equivalent to the XML Schema of Figure 4:

@ Serge Abiteboul, Ioana Manolescu, Philippe Rigaux, Marie-Christine Rousset, Pierre Senellart, 2011; to be published by Cambridge University Press 2011.

```
element book {
  element title { text },
  element author { text },
  element character {
    element name { text },
    element friend-of { text }*,
    element since { xsd:date },
    element qualification { text }
  }*,
  attribute isbn { text }
}
```

It is also possible to define named types and reuse them in multiple places of the schema, like with named XML Schema types. The built-in datatypes of RELAX NG are much less rich than what exists in XML Schema, but RELAX NG offers the possibility of using XML Schema datatypes, as shown in the previous example with the datatype `xsd:date`. RELAX NG is also much more convenient to use when describing unordered content, and does not have the same determinism restrictions as XML Schema.

Schematron. Schematron is a schema language that is built on different principles as XML Schema or RELAX NG. A Schematron schema is an XML document built out of rules, each rule being an arbitrary XPath expression that describes a constraint to be respected. Schematron is not designed as a standalone schema language able to fully describe the structure of a document, but can be used in addition to another schema language to enforce constraints that are hard or impossible to express in this language. Thus, the following Schematron schema ensures that there are as many `a` elements as `b` elements as `c` elements in a whole document:

```
<schema xmlns="http://purl.oclc.org/dsdl/schematron">
  <pattern>
    <rule context="/">
      <assert test="count(//a) = count(//b) and count(//a) = count(//c)">
        Invalid number of characters.
      </assert>
    </rule>
  </pattern>
</schema>
```

Such a constraint on a document is impossible to impose with DTDs, XML schemas, or RELAX NG schemas.

General Programming Languages. In some XML processing contexts, the facilities offered by schema languages for XML are not enough to ensure that a document conforms to a precise type. This might be because the syntax rules are too complex to express in schema languages, or because they deal with syntactic features of XML documents not available in the data model that XML "Validator" use. For instance, the W3C XHTML Validitor checks whether the character set declared in a `<meta http-equiv="Content-Type"/>` tag conforms to the one used in the textual serialization of the document, which is impossible to do using XML

@ Serge Abiteboul, Ioana Manolescu, Philippe Rigaux, Marie-Christine Rousset, Pierre Senellart, 2011; to be published by Cambridge University Press 2011.

schema languages. In these cases, a program in a general programming language, such as C, Java, or Perl, possibly with a DOM or SAX XML parser can be used to enforce these extra constraints. Obviously, one loses the advantages of a declarative approach to XML typing.

4 Typing Graph Data

Using the ID and IDREF attribute types in DTDs, or the `<xs:key />` and `<xs:keyref />` XML Schema elements, it is possible to define XML types where one node of the tree references another node in the tree, moving this way from trees to graphs. In this section, we very briefly present a graph data model and mention alternative approaches to typing graph data.

4.1 Graph Semistructured Data

Prior to XML trees and ID/IDREF links, different models have been proposed for describing graph data. We next mention one.

Definition 4.1 *(Object Exchange Model) An* OEM *is a finite, labeled, rooted graph* (N, E, r) *(simply* (E, r) *when N is understood) where:*

1. *N is a set of nodes;*
2. *E is finite ternary relation subset of* $N \times N \times \mathcal{L}$ *for some set* $\mathcal{L}$ *of labels,* $(E(s, t, l)$ *indicates there is an edge from s to t labeled l);*
3. *r is a node in the graph.*

It should also be stressed that many other graph data models have been considered. In some sense, RDF (see Chapter **??**) is also such a graph data model.

4.2 Graph Bisimulation

A typing of a graph may be seen in some sense as a classification of its nodes, with nodes in a class sharing the same properties. Such a property is that they "point to" (or are "pointed by") nodes in particular classes. For instance, consider Figure 6. A set of nodes forms the class *employee*. From an *employee* node, one can follow *workson*, *leads* and possibly *consults* edges to some nodes that form the class *project*. This is the basis for typing schemes for graph data based on "simulation" and "bisimulation".

A *simulation S* of (E, r) with (E', r') is a relation between the nodes of E and E' such that:

1. $S(r, r')$ and
2. if $S(s, s')$ and $E(s, t, l)$ for some s, s', t, l, then there exists t' with $S(t, t')$ and $E'(s', t', l')$.

The intuition is that we can simulate moves in E by moves in E'.

Given $(E, r), (E', r')$, S is a *bisimulation* if S is a simulation of E with E' and S^{-1} is a simulation of E' with E.

To further see how this relates to typing, take a very complex graph E. We can describe it with a "smaller" graph E' that is a bisimulation of E. There may be several bisimulations for E including more and more details. At one extreme, we have the graph consisting of a single

@ Serge Abiteboul, Ioana Manolescu, Philippe Rigaux, Marie-Christine Rousset, Pierre Senellart, 2011; to be published by Cambridge University Press 2011.

node with a self loop. At the other extreme, we have the graph E itself. This smaller graph E' can be considered as a type for the original graph E. In general, we say that some graph E has type E' if there is a bisimulalation of E and E'.

4.3 Data guides

Another way to type graph data is through *data guides*. Sometimes we are interested only in the paths from the root. This may be useful for instance to provide an interface that allows to navigate in the graph. Consider the OEM Graph of Figure 6. There are paths such as

```
programmer
programmer employee
programmer employee workson
```

It turns out that in general, the set of paths in an OEM graph is a regular language. For instance, for Figure 6, the set of paths can be described by the regular language:

programmer employee leads workson workson?
| statisticien employee leads workson^2 consults?
| employee leads workson workson?
| employee leads workson^2 consults?
| project

A deterministic automaton accepting it is called a *data guide*. A main issue in practice is that the automaton that is obtained "naturally" from the graph is "very" nondeterministic and that the number of states may explode when it is turned into a deterministic automaton.

Observe that the data guide gives some information about the structure of the graph, but only in a limited way: for instance, it does not distinguish between

```
t1 = r ( a ( b ) a ( b d ) )
t2 = r ( a ( b b d ) )
```

that have the same data guide. Note that a document of type t1 has two a elements whereas a document of type $t2$ only has one. Furthermore, suppose the bs have IDs. Then the document of type $t1$ specifies which of the two b elements is related to the d element whereas the second one does not.

5 Further reading

Schema Inference and Static Typing

Several algorithms for inferring schemas from example documents have been proposed, [BNSV10] for DTDs and [BNV07] for XML Schema are recent references that also review other approaches. The problem of static typechecking for XML is reviewed in [Suc02, MSV03].

@ Serge Abiteboul, Ioana Manolescu, Philippe Rigaux, Marie-Christine Rousset, Pierre Senellart, 2011; to be published by Cambridge University Press 2011.

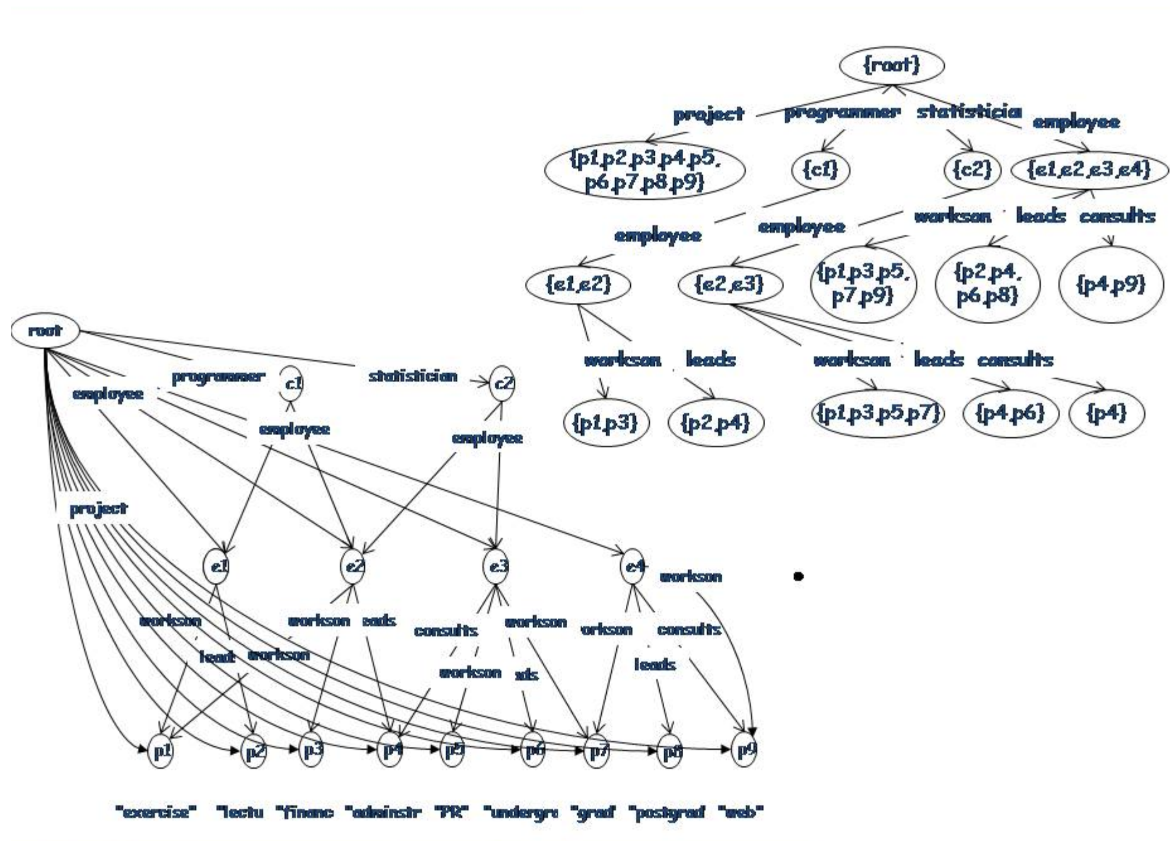

Figure 6: OEM and data guide

@ Serge Abiteboul, Ioana Manolescu, Philippe Rigaux, Marie-Christine Rousset, Pierre Senellart, 2011; to be published by Cambridge University Press 2011.

Automata

The idea that regular tree languages form the proper basis for typing XML was first proposed in [HVP05]. Finite-state automata theory is at the core of computer science. We suggest to the readers not familiar with the topic to further read on it. There are numerous texbooks, e.g., [HMU06]. For everything pertaining to tree automata and regular tree languages, a freely available reference is [CDG+07].

Schema Languages for XML

DTDs are defined in the W3C Recommendation for XML [W3C08]. The specification of XML Schema is split into three documents: an introductory primer [W3C04a], the main part describing the structure of a schema [W3C04b], and a description of the built-in datatypes [W3C04c]. RELAX NG is defined by an OASIS specification [OAS01] and a separate document [OAS02] defines its compact, non-XML, syntax. Both specifications are also integrated in an ISO standard [ISO08a] that is part of the freely available *Document Schema Definition Language* standard. Schematron is defined in another part of this standard [ISO08b]. An interesting comparison of schema languages for XML is presented in [Har03], along with many practical advice on XML document processing.

Typing languages

A very influencial XML-tranformation language, namely XDuce, is presented in [HP03]. Its type checker is based on tree-automata type checking. The language XDuce was then extended to the programming language CDuce [BCF03].

Typing Graph Data

For more details about typing semistructured data in general, including graphs, see [ABS99]. Data guides for semistructured data have been proposed in [GW97]. A query language for the OEM model is proposed in [AQM+97].

6 Exercises

Exercise 6.1 *Show how to construct a right-to-left word automaton that accepts the same language as a given left-to-right word automaton.*

Exercise 6.2 *Show that the languages accepted by nondeterministic bottom-up tree automata and nondeterministic top-down tree automata are the same.*

Exercise 6.3 *The DBLP Computer Science Bibliography,* `http://www.informatik.uni-trier.de/~ley/db/`*, maintained by Michael Ley, provides bibliographic information on major computer science journals and proceedings. DBLP indexes more than one million articles as of 2010. The whole content of the DBLP bibliography is available for download from* `http://dblp.uni-trier.de/xml/` *in an XML format, valid against a DTD available at* `http://www.informatik.uni-trier.de/~ley/db/about/dblp.dtd`*.*

Retrieve the DBLP DTD from the aforementioned URL, and give a document that validates against it and uses all elements and attributes of the DTD.

@ Serge Abiteboul, Ioana Manolescu, Philippe Rigaux, Marie-Christine Rousset, Pierre Senellart, 2011; to be published by Cambridge University Press 2011.

Exercise 6.4 *Consider the following XML document:*

```
<Robots>
   <Robot type="Astromech">
      <Id>R2D2</Id>
      <maker>Petric Engineering</maker>
      <components>
         <processor>42GGHT</processor>
         <store>1.5 zetabytes</store>
       </components>
   </Robot>
   <Robot type="Protocol">
   <Id>C-3PO</Id>
   <maker>Xyleme Inc</maker>
      <components>
         <processor>42GGHT</processor>
         <store>100 exabytes</store>
       </components>
   </Robot>
</Robots>
```

Give a DTD, XML schema, or RELAX NG schema that validate families of robots such as this one.

Exercise 6.5 *Consider the following four sets of documents:*

C_1: `<a><b><c>x</c></b><d><e>y</e></d></a>`
C_2: `<a><b><c>x</c></b><b><e/></b></a>`
C_3: `<a><b><c>x</c></b><b/></a>`
C_4: `<a><b><c/></b><b><c>x</c><c/></b></a>`

where `x` *and* `y` *stand for arbitrary text nodes. Call these sets* C_1, C_2, C_3, C_4.
Questions:

1. *For each* $C_i \in \{C_1, C_2, C_3, C_4\}$*, give a DTD (if one exists) that accepts exactly* C_i*. Otherwise explain briefly what cannot be captured.*
2. *For each* $C_i \in \{C_1, C_2, C_3, C_4\}$*, is there an XML schema that accepts exactly* C_i*? If yes, you do not need to give it. Otherwise, explain briefly what cannot be captured.*
3. *Summarize your results of the first two questions in a table of the form:*

	C_1	C_2	C_3	C_4
DTD	*yes/no*	*yes/no*	*yes/no*	*yes/no*
XML Schema	*yes/no*	*yes/no*	*yes/no*	*yes/no*

4. *Each time you answered "no" in the previous question, give the schema (DTD or XML Schema, according to the case) that is as restrictive as you can and validates* C_i.
5. *Give a DTD that is as restrictive as you can and validates the four sets of documents (i.e.,* $\cup_{i=1}^{4} C_i$*).*
6. *Describe in words (10 lines maximum) an XML Schema as restrictive as you can that validates the four sets of documents (i.e.,* $\cup_{i=1}^{4} C_i$*).*

@ Serge Abiteboul, Ioana Manolescu, Philippe Rigaux, Marie-Christine Rousset, Pierre Senellart, 2011; to be published by Cambridge University Press 2011.

Exercise 6.6 *Consider the trees with nodes labeled f of arity 1 and g of arity 0. Consider the constraint: "all paths from a leaf to the root have even length". Can this constraint be captured by (i) a nondeterministic bottom-up tree automaton, (ii) a deterministic one, (iii) a top-down deterministic tree automaton?*

Same question if f is of arity 2.

Exercise 6.7 *([CDG+07]) Consider the set T of trees with nodes labeled f of arity 2, g of arity 1, and a of arity 0. Define a top-down nondeterministic tree-automaton, a bottom-up one, and a bottom-up deterministic tree-automaton for $G = \{f(a,u), g(v)) \mid u,v \in T\}$. Is it possible to define a top-down deterministic tree automaton for this language?*

[ABS99] S. Abiteboul, P. Buneman, and D. Suciu. *Data on the Web: From Relations to Semistructured Data and XML*. Morgan-Kaufman, 1999.

[AQM+97] S. Abiteboul, D. Quass, J. McHugh, J. Widom, and J. Wiener. The Lorel query language for semistructured data. *Intl. Journal on Digital Libraries*, 1:68–88, 1997.

[BCF03] Véronique Benzaken, Giuseppe Castagna, and Alain Frisch. Cduce: an xml-centric general-purpose language. *SIGPLAN Notices*, 38(9):51–63, 2003.

[BNSV10] Geert Jan Bex, Frank Neven, Thomas Schwentick, and Stijn Vansummeren. Inference of concise regular expressions and DTDs. *ACM Trans. on Database Systems*, 35(2), 2010.

[BNV07] Geert Jan Bex, Frank Neven, and Stijn Vansummeren. Inferring xml schema definitions from xml data. In *Proc. Intl. Conf. on Very Large Databases (VLDB)*, pages 998–1009, 2007.

[CDG+07] H. Comon, M. Dauchet, R. Gilleron, C. Löding, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi. Tree automata techniques and applications. `http://www.grappa.univ-lille3.fr/tata`, 2007.

[GW97] Roy Goldman and Jennifer Widom. Dataguides: Enabling query formulation and optimization in semistructured databases. In *Proc. Intl. Conf. on Very Large Databases (VLDB)*, pages 436–445, 1997.

[Har03] Elliotte Rusty Harold. *Effective XML*. Addison-Wesley, 2003.

[HMU06] J.E. Hopcroft, R. Motwani, and J.D. Ullman. *Introduction to automata theory, languages, and computation*. Addison-Wesley, 2006.

[HP03] Haruo Hosoya and Benjamin C. Pierce. Xduce: A statically typed xml processing language. *ACM Trans. Internet Techn.*, 3(2):117–148, 2003.

[HVP05] Haruo Hosoya, Jerome Vouillon, and Benjamin C. Pierce. Regular expression types for xml. *ACM Trans. Program. Lang. Syst.*, 27(1):46–90, 2005.

[ISO08a] ISO. *ISO/IEC 19757-2: Document Schema Definition Language (DSDL). Part 2: Regular-grammar-based validation. RELAX NG*. International Standards Organization, 2008.

[ISO08b] ISO. *ISO/IEC 19757-3: Document Schema Definition Language (DSDL). Part 3: Rule-based validation. Schematron*. International Standards Organization, 2008.

[MSV03] Tova Milo, Dan Suciu, and Victor Vianu. Typechecking for XML transformers. *Journal of Computer and System Sciences*, 66(1):66–97, 2003.

[OAS01] OASIS. RELAX NG specification. `http://www.relaxng.org/spec-20011203.html`, December 2001.

[OAS02] OASIS. RELAX NG compact syntax. `http://www.relaxng.org/compact-20021121.html`, November 2002.

[Suc02] Dan Suciu. The XML Typechecking Problem. *SIGMOD Record*, 31(1):89–96, 2002.

@ Serge Abiteboul, Ioana Manolescu, Philippe Rigaux, Marie-Christine Rousset, Pierre Senellart, 2011; to be published by Cambridge University Press 2011.

[W3C04a] W3C. XML Schema Part 0: Primer. `http://www.w3.org/TR/xmlschema-0/`, October 2004.

[W3C04b] W3C. XML Schema Part 1: Structures. `http://www.w3.org/TR/xmlschema-1/`, October 2004.

[W3C04c] W3C. XML Schema Part 2: Datatypes. `http://www.w3.org/TR/xmlschema-2/`, October 2004.

[W3C08] W3C. Extensible markup language (XML) 1.0. `http://www.w3.org/TR/REC-xml/`, November 2008.

@ Serge Abiteboul, Ioana Manolescu, Philippe Rigaux, Marie-Christine Rousset, Pierre Senellart, 2011; to be published by Cambridge University Press 2011.