Available online at www.sciencedirect.com
ScienceDirect
Journal of Computer and System Sciences 73 (2007) 289–315
JOURNAL OF COMPUTER AND SYSTEM SCIENCES
www.elsevier.com/locate/jcss

Automata for XML—A survey

Thomas Schwentick

University of Dortmund, Department of Computer Science, 44221 Dortmund, Germany

Received 28 April 2005; received in revised form 28 January 2006

Available online 14 December 2006

Abstract

Automata play an important role for the theoretical foundations of XML data management, but also in tools for various XML processing tasks. This survey article aims to give an overview of fundamental properties of the different kinds of automata used in this area and to relate them to the four key aspects of XML processing: schemas, navigation, querying and transformation.

Keywords: Semistructured data; XML; Automata

1. Introduction

Since the arrival of XML as a data representation language, concepts from formal language theory like regular expressions, grammars and automata have been used for various purposes, e.g., as algorithm models for efficient evaluation of simple queries, as a proof tool, as a tool for static analysis and as an operational model with a clear semantics. Besides automata that read XML documents as strings, called *document automata* in this article, tree automata play an important role, as XML documents have a tree structure and for many applications it is possible to abstract away from text and data details and to model XML data by labeled trees.

For the theory of relational databases (and probably also for their simplicity and widespread use), the close relationship between the query language SQL, the operational relational algebra and the declarative first-order logic was very important. A similarly fruitful interplay seems to exist for XML, this time by the different XML languages designed for various purposes, automata (and algebras) and monadic second-order logic and first-order logic. It should be noted that such a close automata-logic connection has been quite helpful in the area of verification (see [132]).

Although practical XML languages usually do not match exactly with logics and automata, a sufficiently expressive logic can at least offer a framework in which other more restricted languages can be studied. The area of XML schema languages is a particular example.

Generally, logics define robust classes of queries with a clear semantics and they often easily allow to move from one setting to another (e.g., from Boolean to unary to k-ary queries). Automata as an operational model often lead to efficient evaluation and to decidable static analysis.

E-mail address: thomas.schwentick@udo.edu.

0022-0000/$ – see front matter
doi:10.1016/j.jcss.2006.10.003

1.1. Intention and organization

The intention of this article is to give an overview of automata models useful for XML and their basic properties and to survey work on foundations of XML processing that made use of automata in one or the other way. To this end, it basically consists of two parts.

The first part discusses the different kinds of XML processing, exposes the basic algorithmic tasks (Section 2), and provides a background on modeling XML through trees (Section 3) and on automata used for XML (Section 4). The second part covers the four main tasks and their relationship with automata in more detail: schemas (Section 5), navigation (Section 6), transformation (Section 7), and querying (Section 8).

All sections in the second part are organized in a similar fashion: we usually first consider the different (classes of) models and comment on their expressive power. Then we turn to complexity questions, comment on the stream based scenario and discuss some related issues. Each section has its own short summary.

Subsections are usually separated according to the following three different kinds of automata that will be considered:

- *document automata* working on the string representation of documents,
- *sequential tree automata* generalizing the paradigm of finite string automata by allowing one *head* to move along the nodes of a tree, and
- *parallel tree automata* where, basically, many heads move along the tree in vertical dimension (bottom–up or top–down).

1.2. Related work

It should be noted that there is much more work on foundations of XML processing. As this article focuses on automata for XML, it largely ignores investigations which are not directly linked to automata. Still, given the large number of results, it cannot be exhaustive. Other surveys on fundamental aspects of XML can be found, e.g., in [75, 104,105,134].

This article concentrates on the tree model for XML. For work on regular path queries for semistructured data represented by directed graphs which makes frequent use of (string) automata see, e.g., [25,26].

The author apologizes for all aspects and relevant this article is missing, partly due to space restrictions, partly to the ignorance of the author himself.

2. XML processing tasks

There are a lot of processing tasks related to XML. Whereas for relational databases there is, at least in principle, a general-purpose language, SQL, the XML-world is much more diverse. Different types of languages for different kinds of tasks exist and for most tasks several competing language proposals are around. In this article, mainly languages supported by the W3C are considered. We concentrate on the following kinds of languages and their respective application range.

Schema languages: Although the advantage of XML is the possibility to represent data more flexibly than in a relational database, in many applications the structure of documents is in some way restricted. The most common languages to specify the structure of *valid* documents are DTD and XML schema.

Navigation languages: The purpose of a navigation language is to specify positions in an XML document, in general, relative to a given set of positions. Thus, from a database point of view, navigation languages express unary or binary queries. Nevertheless, there is a big difference between navigation and querying. Navigation typically does not produce output but is rather used as a subtask for other kinds of processing. The prime language for navigation is XPath.

Query languages: Queries are used to extract data from XML documents. For this issue many languages have been proposed (e.g., [18]), but most likely the standard language will be XQuery, currently under development by the W3C.

Transformation languages: The distinction between queries and transformations is sometimes not easy. Usually, transformations make use of recursion, most often along the structure of the tree. In contrast, in XQuery recursion is not a core feature but supported only via user-defined functions. Frequently, transformations

Table 1
Algorithmic problems related to XML processing

	Name	Input	Output
Evaluation	VALIDATION	X, d	Does $X \models d$ hold?
	TYPING	X, d	Types of elements w.r.t. d
	NODESELECTION	X, S, q	Set of elements fulfilling q
	QUERYEVALUATION	X, q	Query result defined by q on X
	XFORM	X, T	Result of transforming X w.r.t. T
Static analysis	SATISFIABILITY	d (or q)	Is there X such that $X \models d$ (or $q(X) \neq \emptyset$)
	CONTAINMENT	q_1, q_2	Is $q_1(X) \subseteq q_2(X)$, for every X?
	EQUIVALENCE	q_1, q_2	Is $q_1(X) = q_2(X)$, for every X?
	TYPECHECKING	T (or q), d_1, d_2	Does $T(X) \models d_2$ (or $q(X) \models d_2$) hold, for every X with $X \models d_1$?

X is always a document, d a schema description, q a query (unary in the case of NODESELECTION), T a transformation, and S a set of context nodes. We write $X \models d$ if a document X is valid w.r.t. a schema d.

retain the structure of the input document to a stronger extent. The main language for transformations is XSLT.

Processing of XML data using these kinds of languages leads to many different algorithmic problems. Table 1 lists some important ones that will be discussed in this article. Note the distinction between *evaluation* and *static analysis* problems. Typically, static analysis problems have a much higher complexity than evaluation problems. Nevertheless, as their inputs are (presumably small) queries, transformations, or schemas as opposed to (presumably large) documents, this higher complexity can be often tolerated. For example, even static analysis problems with exponential time complexity are often considered feasible.

These algorithmic problems can be considered in various settings, e.g. in the following.

- The static analysis problems often occur relative to a given schema. E.g., CONTAINMENT relative to a DTD asks, given q_1, q_2 and a DTD d, whether $q_1(X) \subseteq q_2(X)$ for all X with $X \models d$.
- Regarding the computational complexity of evaluation problems there are two points of view. In the *data complexity* perspective the query (or the schema) is considered as fixed, whereas in the *combined complexity* perspective it is part of the input.
- All problems can also be considered in a streaming context, where the elements appear in *document order* and have to be processed immediately after they are seen.

3. XML and trees

In first place, an XML document is a sequence of symbols, i.e., a string. The structure of *valid* XML documents is described in [136] by a combination of, basically, an extended context-free grammar with additional constraints.

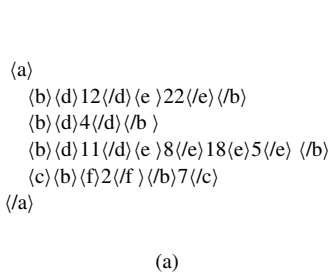
Figure 1(a) shows an example of a (toy) XML document.

As they are strings, XML documents can, in principle, be handled by string automata (called *document automata* below). Nevertheless, for reasons discussed below, most of the investigations covered in this survey exploit the intrinsic tree structure of XML documents and employ *tree automata* instead of string automata. Thus, we discuss in this section several approaches to represent XML documents as trees.

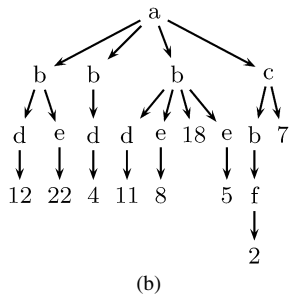
3.1. XML documents as trees

The most direct way of representing an XML document as a tree is illustrated in Fig. 1(b).

- Elements and text strings of the document become nodes of the tree.
- Whenever an element or a text string is directly contained in another element, there is an edge from the node representing the containing element to the one representing the content. Thus, nodes representing character data are always leaves of the tree (but not vice versa).



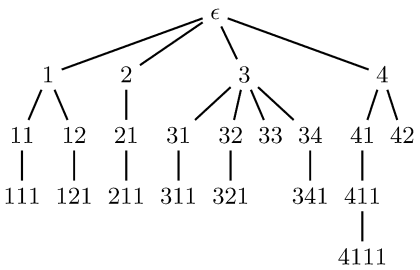
(a)



(b)

a(b(d(12),e(22)),b(d(4)),b(d(11),e(8),18,e(5)),c(b(f(2)),7))

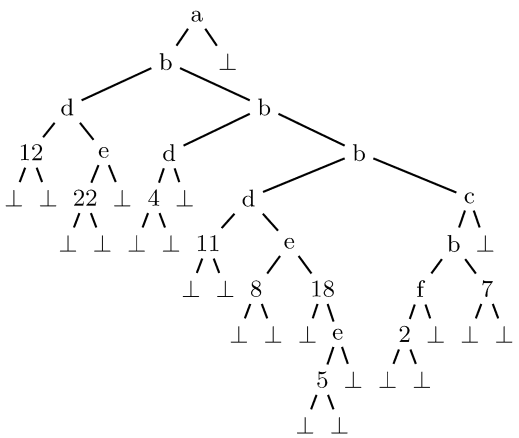
(c)



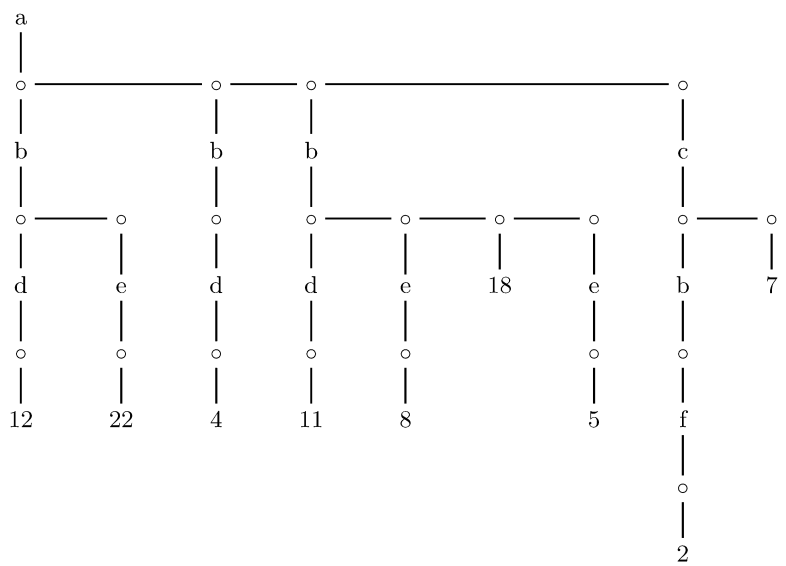
(d)

$l(\epsilon) = a$
$l(1) = b$
$l(11) = d$
$l(111) = 12$
$\vdots$

(e)



(f)



(g)

Fig. 1. Example document (a), as a tree (b), as a term (c), the underlying tree domain (d), its label function (e), its binary encoding via first-child-next-sibling (f), as in [92] (g) (⊥-nodes omitted).

- Additionally, the children of a node are ordered from left to right, corresponding to the order in the document. Therefore, as opposed to usual graph presentations, the order in which the children of a node are depicted is important.
- Finally, in the tree representation, element nodes are *labeled* by the respective element name.

In trees resulting from XML documents in this way, the number of children of a node has no a priori limit. Such trees are called[1] **unranked trees** as opposed to **ranked trees**, in which the number of children of a node is bounded by some fixed constant.

A different approach is to represent XML documents as *terms*. This approach is best defined recursively. A text string s is represented by s itself. An element with tag ⟨a⟩, whose direct subelements are represented by $t_1, \ldots, t_n$ (in that order), is represented by $a(t_1, \ldots, t_n)$. Hence, a becomes a function symbol. Of course, the difference between viewing XML documents as terms or as (graph-theoretic) trees is not very big. Although, there are some settings in which the term approach is quite natural, we will pursue in this article only the graph-based point of view.

As this survey will mainly consider on investigations which concentrate on the structural content of XML documents, the example document does not contain any attributes. Nevertheless, it is not hard to incorporate the representation of attribute values into trees. Whenever an element e has attribute specifications, the node representing e has one child labeled with the attribute name, for each attribute specification. Attribute values can in turn be represented as children of their respective attribute name nodes. Some care is needed though, as the order of attributes is not significant in XML. Thus, when processing such trees with automata, one has to make sure that the order of attribute name nodes does not affect the result.

In [122] a pointer-based encoding of XML documents is considered which uses, for each node, a label and pointers to its parent, first child, right sibling and left sibling, respectively.

Sometimes we will relate the expressive power of automata with classes of logical formulas. To this end, trees are represented as finite structures in which the underlying universe consists of the set of nodes of the tree. Which predicates can be used by a formula depends on the particular setting. Usually, formulas can test whether a node has a specific label, i.e., an atomic formula $a(x)$ evaluates to true, if the node bound to x has label a. Concerning the structure of the tree, in some settings, the child and the next-sibling predicate are available, i.e., formulas can use atoms $\texttt{child}(x, y)$ and $\texttt{nextsibling}(x, y)$ testing whether (the node bound to) y is a child of x and whether y is the right neighbor of x, respectively. In other cases also the `descendant` and the `followingsibling` predicate is available. For more information on logics for XML-trees we refer to [75].

3.2. A formal representation of XML-trees

When it comes to a formal definition of abstract XML trees it is convenient to consider the tree structure on the one hand and the labels (tags), character data and attribute values, on the other hand separately.

Very often the tree structure is represented as follows. Let $\mathbb{N}^*$ denote the set of finite sequences over the set $\mathbb{N}$ of natural numbers. Although $\mathbb{N}$ is not a finite set, the elements of $\mathbb{N}^*$ are considered as a kind of strings, in particular string terminology is used like *prefix* and *suffix*. A (finite) **tree domain** D is a finite set of elements of $\mathbb{N}^*$ with the following two closure properties.

(1) If $v \in D$ and u is a prefix of v then $u \in D$;
(2) If $vi \in D$, $i \in \mathbb{N}$ and $j < i$ then also $vj \in D$.

Let Σ be an alphabet (of labels/tags) and Γ be an alphabet over which text strings are formed. A **tree** $t = (D, l)$ consists of a tree domain D and a mapping $l : D \to \Sigma \cup \Gamma^*$ mapping each node to its label/tag or text string. The value $l(v)$ can be in Γ^* only if v is a leaf. The tree domain of our example tree and part of the function l are depicted in Fig. 1(d) and (e).

A sequence of trees is often called a *hedge* to distinguish it from an (unordered) forest. Murata [95] attributes this naming to [33].

[1] Strictly speaking, this terminology only makes sense for sets of trees as a single tree always has a degree bound.

In settings, in which character data and attributes are irrelevant, the labeling map l takes only values from Σ. The resulting trees are called **Σ-trees**.

It should be noted here that, for many purposes, character data and attributes can be represented by Σ-trees. E.g., if a query asks for all nodes containing a given attribute value "a," then Σ can be chosen as the set of labels of the form (σ, b), where σ is the usual label of a node and $b \in \{0, 1\}$ indicates whether the attribute value of the node is "a."

3.3. Encoding unranked trees as binary trees

The theory of tree automata has first been developed for ranked (typically binary) trees. It is therefore tempting to encode XML trees into binary trees and to use traditional binary tree automata. A very natural encoding is as follows (cf. [72]). Each non-leaf node has one outgoing edge to its first child and each node that is not the rightmost child of some other node (and not the root) has an edge to its right sibling. We refer to this as the *first-child-next-sibling encoding*. Figure 1(f) shows the resulting binary tree for our example document.

But this is by no means the only way of encoding an unranked tree into a binary one. One of the disadvantages of the first-child-next-sibling encoding is that it makes it complicated to identify vertical paths of the original tree. For this reason, [92] used the following encoding. First, for each non-root node a new parent node marked by $\circ$ is added as a dummy node. Then the first-child-next-sibling construction is applied. In addition, for leaf nodes and rightmost children extra nodes marked $\circ$ are added indicating that there is no child or right sibling, respectively. Figure 1(g) gives an example.

Yet another encoding is used in [48]. It inserts between a node and its children an almost complete binary tree of intermediate nodes resulting in an overall binary tree which reflects the original structure more closely. The depth of the resulting binary tree is at most increased by a factor which is logarithmic in the maximal rank of a node. In contrast, the previous encodings might result in binary trees with depth depending linearly on the number of nodes of the original tree.

In [27] a further encoding is introduced which can be used to reduce unranked bottom–up tree automata to ranked ones.

4. Automata

As mentioned in the introduction, we consider three main categories of automata, *parallel tree automata*, *sequential tree automata* and *document automata*. We define these automata models in the three subsections of this section and survey results about their expressive power and the complexity of basic algorithmic tasks.

As tree automata were first studied for ranked trees, the respective subsections first review results for ranked tree automata and then discuss the necessary adaptations for unranked tree automata.

When moving from strings to trees, it is helpful to view strings as unary trees, i.e., as trees in which each non-leaf node has exactly one child. A (one-way) string automaton then proceeds from the root to the unique leaf (or vice versa). Two aspects of such an automaton are relevant for the generalization to trees: the automaton has *one head* and it proceeds strictly vertically in *only one direction* (top–down or bottom–up).

Sequential tree automata, discussed in Subsection 4.2, retain the one-head paradigm. Parallel automata, discussed next, process in only one direction but in order to do so they make use of many heads.

By and large, parallel tree automata are used in XML theory for schema languages and node-selecting queries whereas sequential tree automata play a role in navigation and, as the basic ingredient of pebble tree transducers, in transformation. Whether automata can also be useful for general querying has still to be explored.

4.1. Parallel tree automata

We first survey some classical work on *ranked* tree automata. For simplicity we only consider automata for *binary* trees. The generalization for ranked trees of arbitrary (bounded) arity, which we discuss afterwards, is in most cases straightforward.

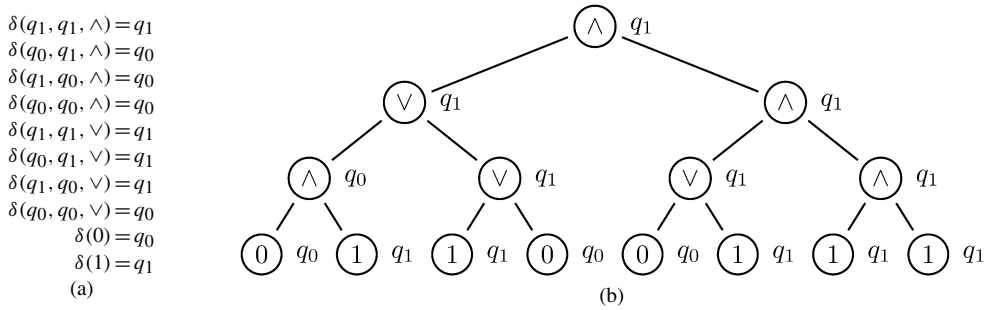


Fig. 2. Deterministic bottom–up automaton (a) and tree-structured Boolean circuit with the states that the automaton takes (b).

4.1.1. Ranked parallel tree automata

As in the string case, one can consider deterministic and non-deterministic tree automata and they can work in a bottom–up or top–down fashion. This leads to the 4 main types of parallel tree automata. We start with the conceptually most simple model.

A **deterministic bottom–up binary tree automaton**[2] **(DBTA)** $\mathcal{A} = (Q, \Sigma, \delta, F)$, consists of a finite set Q of states, a finite alphabet Σ, a set $F \subseteq Q$ of final states, and a transition function $\delta : \bigcup_{i=0}^{2} Q^i \times \Sigma \to Q$. Here, Q^2 denotes $Q \times Q$. The semantics of an automaton $\mathcal{A}$ on a binary Σ-tree t is defined by inductively assigning to each node v of t a state $q(v)$ as follows.

- If v is a leaf then $q(v)$ is $\delta(l(v))$.
- If v is a node with one child u then $q(v)$ is $\delta(q(u), l(v))$.
- If v is a node with left child u_1 and right child u_2 then $q(v)$ is $\delta(q(u_1), q(u_2), l(v))$.

A tree t with root node v is accepted by $\mathcal{A}$ if $q(v) \in F$.

An example automaton which evaluates tree-structured Boolean circuits is depicted in Fig. 2. It has only two states, q_0, q_1. The idea is that a node gets state q_1 just in case its subtree evaluates to TRUE.

In a **non-deterministic bottom–up binary tree automaton (DBTA)**, there is, in general, more than one way to assign states to nodes. Consequently, δ is a relation rather than a function, i.e., $\delta \subseteq \bigcup_{i=0}^{2} Q^i \times \Sigma \times Q$.

A function q assigning to each node v a state $q(v)$ is a ***run*** of the automaton if

- for each leaf node v, $(l(v), q(v)) \in \delta$,
- for each node v with one child u, $(q(u), l(v), q(v)) \in \delta$, and
- for each node v with left child u_1 and right child u_2, $(q(u_1), q(u_2), l(v), q(v)) \in \delta$.

A run is **accepting** if the root has a state in F.

Non-deterministic *top–down* binary tree automata are defined very similarly as bottom–up automata but they will not be important for the purpose of this article. The following theorem is classical.

Theorem 4.1. *For a set L of binary Σ-trees the following are equivalent.*

(a) *L is accepted by a deterministic bottom–up automaton.*
(b) *L is accepted by a non-deterministic bottom–up automaton.*
(c) *L is accepted by a non-deterministic top–down automaton.*

Proof *(Idea).* The equivalence between (a) and (b) is obtained by a powerset construction very similarly to the case of string automata. The equivalence between (b) and (c) is straightforward. □

[2] Whether an automaton is ranked or not will usually be clear from the context and is therefore not represented in the abbreviation DBTA.

The automata considered so far thus define a robust class of tree languages, the (ranked) **regular tree languages**.

A further important characterization of this class is by closed[3] formulas of monadic second-order (MSO) logic [38,128]. Besides the usual first-order quantification of nodes, MSO-formulas can quantify over sets of nodes.

Theorem 4.2. [38,128] *A set L of binary Σ-trees is regular if and only if it is the set of models of a closed MSO formula.*

That MSO-formulas can "simulate" automata is quite straightforward. For each state, the set of nodes assuming this state can be existentially "guessed." Then, a first order formula can express that these sets induce an accepting run of the automaton. The other direction of the proof is more involved. Note that the theorem holds no matter whether only `child` and `nextsibling` or also `descendant` and `followingsibling` are available.

The notion of deterministic top–down automata is more delicate. We start with a weak model.

A **deterministic top–down binary tree automaton (DTTA)** $\mathcal{A} = (Q, \Sigma, q_0, \delta, F)$ has a distinguished initial state q_0, a set $F \subseteq Q \times \Sigma$ and a pair $\delta = (\delta_1, \delta_2)$ of transition functions, $\delta_1 : Q \times \Sigma \to Q$ and $\delta_2 : Q \times \Sigma \to Q^2$. It induces a state assignment q to a tree t as follows.

- The root v gets state q_0.
- If a node v has exactly one child u then $q(u)$ is $\delta_1(q(v), l(v))$.
- If a node v has two children u_1, u_2 then $(q(u_1), q(u_2))$ is $\delta_2(q(v), l(v))$.

Finally, a tree t is accepted by $\mathcal{A}$, if for all leaves v, $(q(v), l(v)) \in F$.

Defined in this way, deterministic top–down automata cannot express the set of trees that contain at least one node with a given label a. Furthermore, the class of tree languages definable by deterministic top–down automata is neither closed under complementation nor under intersection. On the other hand, the obvious alternative to define acceptance in an existential fashion ("at least one leaf accepts") cannot express that all nodes have label a and lacks closure under complementation and union.

Part of this weakness is due to the fact that such automata have no means to combine information collected along different paths. In fact, it can be shown that acceptance of a deterministic top–down automaton only depends on the set of vertical paths of the tree (where each node carries a label and information about whether it is a left or right child) [34].[4]

A much stronger top–down model with more enjoyable closure properties was defined in [69]. There, the acceptance condition is induced by a regular string language L_F over the alphabet $Q \times \Sigma$. A tree t is accepted if the sequence of pairs $(q(v_1), l(v_1)), \ldots, (q(v_m), l(v_m))$ is in L_F, where $v_1, \ldots, v_m$ are the leaf nodes of t, numbered from left to right.

An exhaustive source for information on ranked parallel tree automata is [32].

4.1.2. Unranked parallel tree automata

When moving from ranked to unranked trees there are basically two options to deal with automata. The first is to encode unranked trees by binary trees and to apply ranked automata. This approach has been taken in several papers (e.g., in [27,127] and implicitly in [55]). It turned out that it leads to the same notion of unranked regular tree languages as the second approach which we describe next. This second approach is often more intuitive as it deals with XML trees more directly. Nevertheless, when it comes to implementations, binary encodings are very useful.

For ranked automata, δ is a finite function or relation. This is no longer the case for (reasonable definitions of) tree automata for unranked trees. Hence, a possibly infinite set of transitions has to be represented in a finite way. As the power of the resulting model should be "regular" it is an obvious choice to use regular string languages for this purpose.

There are many equivalent ways to formalize this basic idea. We follow [102]. For a **non-deterministic bottom–up unranked tree automaton** $\mathcal{A} = (Q, \Sigma, \delta, F)$, Q, Σ and F are as before. For each $q \in Q, \sigma \in \Sigma$, $\delta(q, \sigma)$ is a regular language over Q. The semantics is defined as follows: If a node v with label σ has children $u_1, \ldots, u_m$ and, for each i,

[3] A formula is closed if it does not have any free variables.

[4] In [34] this is shown in the context of unranked trees.

q_i is the state assigned to u_i then the state q can be assigned to v if $q_1 \cdots q_m \in \delta(q, \sigma)$. $\mathcal{A}$ is **deterministic** if, for each σ the languages $\delta(q, \sigma)$ are pairwise disjoint.

Non-deterministic top–down unranked automata can be defined analogously. It is not hard to see that the analogues of Theorem 4.1 and Theorem 4.2 hold for unranked tree automata as well.

Theorem 4.3. [21,94,102] *For a set L of unranked Σ-trees the following statements are equivalent.*

(a) *L is accepted by a deterministic bottom–up automaton.*
(b) *L is accepted by a non-deterministic bottom–up automaton.*
(c) *L is accepted by a non-deterministic top–down automaton.*
(d) *L is the set of models of a closed MSO formula.*

Thus, it is natural to refer to this class of languages as the *regular unranked tree languages*.

Again, the case of deterministic top–down (unranked) automata is more complicated (cf. [34,86]).

4.2. Sequential tree automata

In contrast to parallel tree automata, sequential tree automata have only one single head which moves along the edges of a tree. Again, we consider automata for ranked (and even binary) trees first.

4.2.1. Ranked sequential tree automata

The basic sequential tree automata model is that of tree-walking automata (TWAs) [1]. A more powerful model, which we consider afterwards, is obtained by adding pebbles to TWAs.

A **deterministic tree-walking automaton (DTWA)** has a single head which is moved along the edges of a tree, from node to node. Depending on the current node label, the current state and whether the node is the root, a leaf, a left or a right child, the head moves to the parent or to one of the children and takes a new state. The computation starts in the root in a distinguished initial state q_0 and ends if an accepting or rejecting state is reached. Note that it might happen that a computation does not terminate. **Non-deterministic tree-walking automata (NTWA)** have, in general, more than one transition that they can take in a given situation.

Although tree-walking automata are not limited to traverse the tree in one direction, it is not too hard to see that they only accept regular tree languages. In fact, their expressive power seems to be quite limited, at first sight. Nevertheless, a closer inspection shows that they can recognize some non-trivial properties of trees.

As an example, there is a DTWA which tests whether a tree-structured Boolean circuit evaluates to TRUE. To this end, the automaton traverses the tree in depth-first left-to-right order. The crucial idea is that if it comes back from a FALSE-valued left subtree of an $\wedge$-node it does not need to consider the right subtree of that node, as the node evaluates to FALSE anyway. Similarly, it does not need to enter the right subtree if it comes back from a TRUE-valued left subtree of an $\vee$-node. In turn, this implies that if the automaton comes back from a right subtree, it always can infer the value of the left subtree. This value is FALSE if the node is an $\vee$-gate and TRUE otherwise.

Furthermore, as shown[5] in [23], DTWAs can recognize all tree languages of the form $PC(L)$, where L is a regular string language and $PC(L)$ denotes all trees whose root-to-leaf paths have only label sequences in L.

Despite these examples, it seems quite obvious that neither deterministic nor non-deterministic tree-walking automata can accept all regular tree languages. Nevertheless, this had merely been a conjecture for many years and only partial results had been known [108] until the issue was recently resolved.

Theorem 4.4. [14,15]

(a) *There is a tree language accepted by a non-deterministic tree-walking automaton but not by any deterministic tree-walking automaton.*
(b) *There is a regular tree language not accepted by any non-deterministic tree-walking automaton.*

[5] In [23], the result is shown in the context of caterpillars which will be discussed below.

This theorem is relevant for the context of XML as TWAs are the basis for other formalisms like (equivalent) caterpillars and (stronger) pebble automata, to be defined below. As will be explained in Section 7, pebble automata are used as an ingredient for transducers, thus it is important to understand their expressive power. It is conjectured that they also fall short of capturing all regular tree languages.

The expressive power of tree-walking automata can be enlarged in various ways. One option is to add a push-down which can be used only in a limited way (basically: *depth-synchronous*, as it will be defined for document automata below). This model captures the regular tree languages and has a close relationship to the top–down tree transducers considered in Section 7 (cf. [46,130]) by adding *pebbles*. These **pebble tree automata** are quite useful in the XML context. Intuitively, when a pebble tree automaton visits a node it can place a pebble on that node and it can remove pebbles from nodes. The number of pebbles is fixed, for each automaton. Without restricting the placement of pebbles the resulting model would be a kind of multihead automaton with the ability to express all LOGSPACE properties on trees. Therefore, the placement and lifting of pebbles is restricted to follow a stack discipline, i.e., the pebbles are numbered from 1 to k and at each point in time, if i is the highest number of a pebble on a node, then only pebble i can be removed or pebble $i+1$ can be placed at the current node.

Within this framework there are some subtle differences between the exact procedure which a pebble i lifted (cf. [40]):

(1) Pebble i can be lifted no matter whether the head is at its position, and the position of the head is not affected.
(2) The pebble can be lifted only if the head is at the same node and the head remains at that node.
(3) Pebble i can be lifted only if the head is at the same node and the head is afterwards instantly moved to the position of pebble $i-1$ (unless $i=1$).

It can be shown that all three kinds of pebble tree automata only accept regular tree languages [42]. It has been conjectured in [42] that they do not capture all regular tree languages but this question is still open. There are extensions of pebble automata by *marbles* or *set pebbles* which capture the regular tree languages [42,43].

Non-deterministic pebble tree automata (of the first kind) capture exactly those tree languages that can be characterized by first-order formulas extended with positive applications of the unary transitive-closure operator [40]. A corresponding characterization for deterministic pebble automata is given in terms of the unary deterministic transitive-closure operator. Therefore, whether pebble tree automata are weaker than parallel tree automata is the same question as whether MSO-logic is more powerful on trees than first-order logic with unary positive transitive closure. It should be noted that on strings, the expressive power of these logics and all the automata models coincide.

4.2.2. Unranked sequential tree automata

In principle, the paradigm of one head moving from node to node can be easily generalized to unranked trees. From a node v, such an automaton can do the most natural steps: it can go up to the parent of v, down to its first child, down to its last child, to its right sibling and to its left sibling, of course only if the respective node exists. To this end, the automaton should be able to check whether v is the root, a leaf, a left-most or a right-most child. Nevertheless, defined in this way, tree-walking automata suffer from an additional weakness (on top of the weakness in comparison with parallel automata): Consider a node u which is one of the middle children of a node v with many children. If the head of an automaton is at u it might be difficult if not impossible to find out the label of v and come back to u again (as it only can go via the left-most or right-most child of v which might be far from u). This is in contrast to the transitions in parallel automata which usually depend on the label of the parent of a node. It might therefore be sensible to allow a tree-walking automaton to check the label of its parent while staying at u. It seems that the expressive power of this extended kind of tree-walking automata on unranked trees has not been explored so far, although it is clear that they cannot accept all regular tree languages due to the results in [14,15].

Pebble automata can be defined for unranked trees in a similar fashion. Actually, the logical characterizations for ranked pebble automata directly generalize to unranked trees.

There is an alternative formulation of tree-walking automata for unranked trees, **caterpillar expressions** (short: *caterpillars*), introduced in [23]. A caterpillar is a regular expression over an alphabet $\Sigma \cup \Delta$, where Σ is the alphabet of labels of the tree and Δ is the alphabet of directions and tests, $\Delta = \{\text{up, left, right, first, last, isFirst, isLast, isLeaf, isRoot}\}$. A caterpillar C defines, for each tree, a set of nodes (the *context set*). As an example, the context set of the

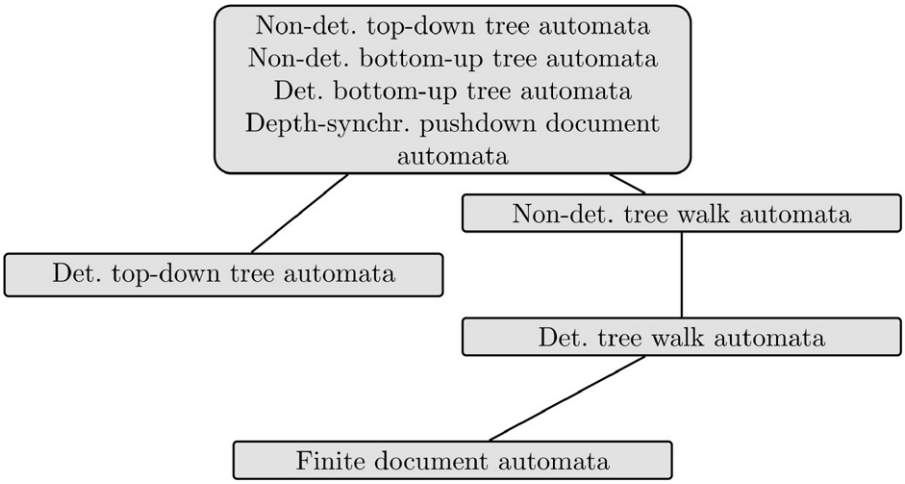


Fig. 3. Expressive power of some tree automata models.

expression $a \cdot \text{up} \cdot \text{isLeft} \cdot \text{up} \cdot \text{isRoot}$ contains all a-labeled children of the first child of the root. A caterpillar C also induces a tree language $L(C)$ containing all trees with a non-empty context set.

Following the same lines as in the case of strings, it can be shown that, with respect to the definition of tree languages, caterpillars and non-deterministic tree-walking automata (without the ability to check the parent-label) are equivalent.

4.3. Document automata

Although it is natural to model XML documents by trees and to apply tree automata to these trees, from a processing point of view *document automata*, i.e., string automata which read the documents directly, are quite natural. When the set of possible opening and closing tags is given by an alphabet Σ and the occurring strings use only symbols from an alphabet Γ we have to consider automata with alphabet $\Sigma \cup \Gamma$. This approach is especially appealing if XML data arrives in a streaming fashion.

In order to be able to keep track of the structure of opening and closing tags, usually pushdown automata are used to process streaming XML data. Nevertheless, in [123] finite automata have been considered in the context of validation.

Push-down automata can recognize (the document representations of) all regular tree languages, but their expressive power is strictly stronger than that. By restricting them to push a symbol whenever they read an opening tag and to pop a symbol when they read a closing tag, they capture exactly the regular tree languages. We refer to such automata as *depth-synchronous pushdown automata*. They are a restricted kind of *visibly pushdown automata* [4].

It should be noted that in settings where the depth of all considered trees is bounded by some fixed constant, finite string automata have the same expressive power (w.r.t. acceptance of languages) as parallel tree automata.

Figure 3 shows the relative expressive power of most of the automata models mentioned in this section.

4.4. Complexity of algorithmic problems related to tree automata

The basic algorithmic problems related to tree automata are the following.

- MEMBERSHIP: Given a tree t and an automaton $\mathcal{A}$, check whether $\mathcal{A}$ accepts t.
- NON-EMPTINESS: Given an automaton $\mathcal{A}$, is there a tree t accepted by $\mathcal{A}$?
- CONTAINMENT: Given automata $\mathcal{A}_1, \mathcal{A}_2$, is every tree accepted by $\mathcal{A}_1$ also accepted by $\mathcal{A}_2$?
- EQUIVALENCE: Given automata $\mathcal{A}_1, \mathcal{A}_2$, do they accept exactly the same trees?

The complexity of the MEMBERSHIP problem has been studied in [76,122].

Table 2
Time bounds and structural complexity for the MEMBERSHIP problem (combined complexity) [76,122]

Model	Time complexity	Structural complexity
Det. top–down TA	$O(\|\mathcal{A}\|\|t\|)$	LOGSPACE
Det. bottom–up TA	$O(\|\mathcal{A}\|\|t\|)$	LOGDCFL
Non-det. bottom–up TA	$O(\|\mathcal{A}\|^2\|t\|)$	LOGCFL
Non-det. top–down TA	$O(\|\mathcal{A}\|^2\|t\|)$	LOGCFL
Det. TWA	$O(\|\mathcal{A}\|^2\|t\|)$	LOGSPACE
Non-det. TWA	$O(\|\mathcal{A}\|^3\|t\|)$	NLOGSPACE

The complexity results are all completeness results w.r.t. suitable reductions.

If the automaton is considered fixed (data complexity) the problem is complete[6] for NC^1 [76]. If the automaton is part of the input the exact complexity depends on the automaton model. Table 2 lists the complexity results as well as upper bounds for the time complexity.

Whereas for string automata the NON-EMPTINESS problem is complete for NL, for (parallel) tree automata it is complete for P [133]. The author is not aware of any results concerning the complexity of NON-EMPTINESS or CONTAINMENT for tree-walking automata.

For the complexity of the CONTAINMENT problem there is a huge (provable!) difference between non-deterministic and deterministic bottom–up parallel automata. For deterministic bottom–up automata the problem is complete for P. The upper bound is obtained by computing the product of the complement automaton of $\mathcal{A}_2$ with $\mathcal{A}_1$. For non-deterministic automata it is complete for EXPTIME [124].

The complexities are in all cases the same for ranked and for unranked tree automata.

4.5. Related issues

For each ranked regular tree language, there exists a unique minimal deterministic bottom–up tree automaton [112]. For top–down automata corresponding results depend again on the exact model [50,112]. For unranked trees, the existence of a unique minimal automaton depends on the precise definition of the model [34,86]

It is a long standing open problem whether, given a tree automaton, one can decide effectively if its corresponding tree language can be expressed in first-order logic. This is answered affirmatively in [8], for the case where the formulas can only use the `child` and the `nextsibling` predicate. Whether it also holds in the presence of the `descendant` and the `followingsibling` predicate is still open (see also [75]). On strings, the corresponding problem is decidable (cf. [129]).

Cascading sequences of automata have been investigated in [13]. The idea is that several automata traverse the tree, each of which can read the state in which the previous one visited a node. By restricting the structure of the single automata one can capture first-order logic and chain logic (with quantification over sets of nodes on a vertical path) on trees.

4.6. Section summary

Tree automata for ranked trees can be roughly divided into two groups, parallel and sequential. Most parallel models are equivalent with respect to their language recognition power, giving rise to the regular tree languages. Only deterministic top–down automata are weaker and their expressive power depends on the particular definition. Sequential automata are typically less expressive than parallel automata. The complexity of the MEMBERSHIP problem is low for all models and the difference between parallel and sequential automata is not big. NON-EMPTINESS for parallel automata is tractable, CONTAINMENT only for deterministic ones.

Moving from ranked to unranked trees is either possible by encoding unranked trees as binary trees or by extending the definition of tree automata to unranked trees. Most of the basic results carry over to unranked tree automata, in particular there is a robust class of regular tree languages.

For XML processing also document automata are relevant.

[6] For the exact reducibility notions we refer to the original papers to keep the exposition less technical.

$$\begin{array}{ll} a \to b^*c & d \to x \\ b \to (d+\epsilon)(e+x)^* & e \to x \\ c \to bx^* & f \to x \end{array}$$

Fig. 4. Extended context-free grammar for the example document in Fig. 1. The symbol x represents arbitrary text.

5. Schemas

Schema languages for XML are closely related to the theory of formal languages in general and, in particular, to automata. In this section, we consider three classes of schema languages of different expressive power. First, we have a look at (DTDs), the schema language included in the XML specification. Second, we consider the much larger class of regular tree languages which provides a general framework to study schema languages. Then, we examine languages of intermediate expressive power like XML schema.

After discussing the complexity of some algorithmic tasks for these languages, as an orthogonal topic, we consider validation of streaming XML data. Finally, we discuss some further issues.

As we are mainly interested in automata, we concentrate in this survey on the structural aspects of schema languages and largely ignore the matter of integrity constraints.

5.1. A weak class of schema languages: DTDs

The simplest and most frequently used way of describing the schema of XML documents, already specified in the XML standard, is via DTDs (*document type descriptions*). As it is common, we model DTDs by extended context-free grammars. In an *extended context-free grammar* the right-hand side of each rule is a regular expression as opposed to a sequence of terminals and non-terminals. Note that an element can become a leaf node just in case the right-hand side of its rule describes a language containing the empty string.

An example of an extended context-free grammar, corresponding to the example document of Fig. 1 is given in Fig. 4.

DTDs correspond to *local tree languages* [97]. These languages can be characterized by a closure property: a tree language L is local if and only if, whenever v, v' are nodes of trees $t, t' \in L$ with the same label, the tree resulting from replacing the subtree t_v of t rooted at v by $t'_{v'}$ is also in L [115].

To ensure efficient validation w.r.t. DTDs, the W3C recommendation enforces a special form of regular expressions in DTDs: "It is an error if an element in the document can match more than one occurrence of an element type in the content model [without looking ahead]." This requirement has been formalized in [22] by the notion of *one-unambiguous* regular expressions which we discuss next.

5.1.1. One-unambiguity

Given a regular expression E, let E' be obtained by indexing the symbols in E consecutively from left to right. E.g., if $E = bc + bd$ then $E' = b_1c_2 + b_3d_4$. E is **one-unambiguous** if the following condition holds. There are no two words $u\sigma_i v$ and $u\sigma_j w$ in $L(E')$ where u, v, w are strings, σ is a symbol and $i \neq j$.

Taking u as the empty string ϵ, it can be seen that the example expression E is not one-unambiguous. The expression $b(c + d)$ defining the same language is one-unambiguous, though.

Whether a regular expression is one-unambiguous can be tested by checking whether its Glushkov automaton [51] is deterministic. The *Glushkov automaton* has the symbols of E as states (plus an initial state) and the transitions are defined in a straightforward way. In particular, for one-unambiguous regular expressions, a deterministic automaton can be obtained in quadratic time.

5.1.2. DTDs and automata

Whether a tree is valid w.r.t. to a DTD can be tested by most automata models we considered so far, even by (some model of) deterministic top–down automata and by deterministic tree-walking automata. The latter even holds for the TWA-model in which the automaton is not aware of the label of the parent of a node. To this end, the DTWA does basically a depth-first left-to-right pass through the tree and checks, for every node, that the sequence of children is valid.

	Dealer → UsedCars NewCars	μ(Dealer) = Dealer
Dealer → UsedCars NewCars	UsedCars → adUsed*	μ(UsedCars) = UsedCars
UsedCars → ad*	NewCars → adNew*	μ(NewCars) = NewCars
NewCars → ad*	adUsed → model year	μ(adUsed) = ad
ad → (model year) + model	adNew → model	μ(adNew) = ad
(a)	(b)	

Fig. 5. DTD (a) and extended DTD (b) for car dealer example.

5.2. A stronger class: Regular schema languages

DTDs are considered as too weak for many applications. One of the main deficiencies is that element names are always interpreted in the same way, regardless of the context. As a classical example (from [115]) consider a car dealer who sells new as well as used cars. A (toy) DTD for his store might look as depicted in Fig. 5(a). Intuitively, an ad should contain information about the construction year if and only if a car is used. But DTDs are to weak to specify this.

As a solution to this problem, [115] proposed an extension which we will call[7] *extended DTDs*. The basic idea of extended DTDs is that with each element in a document a type is associated. In the example, there are two types of ad elements, adUsed and adNew. An extended DTD specifies which sequences of types can occur inside an element of a certain type and which types are associated to which element name (each type is associated with a unique element name!). More formally, an extended DTD consists of a DTD over the alphabet of types and a mapping μ from types to element names. Figure 5(b) displays an extended DTD for the car dealer example. As will be discussed below, XML schema supports a restricted form of extended DTDs.

To test whether a document is valid w.r.t. an extended DTD, one has to associate a type with each node and to check whether the children of each node fulfill the corresponding regular expressions. The attentive reader might already have noticed that we have seen this mechanism before, under a different name. Actually, the types can be considered as states of a non-deterministic automaton and the regular expressions as specifications of possible transitions. Hence, for every extended DTD there is an equivalent non-deterministic parallel tree automaton accepting the same language. It is not hard to see that the converse is also true. Therefore, a language of unranked trees can be described by an extended DTD if and only if it is regular.

Thus, regular tree languages offer an elegant framework for the study of schema languages, even for restricted languages as those discussed in the next subsection.

Regular tree languages as a framework for XML schema languages have also been considered in [28,97].

Examples of schema languages which capture all regular tree languages are XDuce [65] and Relax NG [31].

For extended DTDs a new algorithmic problem becomes relevant, the TYPING problem. Given a document and an EDTD it asks for an assignment of types to the elements of the document which is valid with respect to the EDTD. It should be noted that TYPING is essentially a (consistent) combination of unary queries.

5.3. Intermediate classes of schema languages

As DTDs are a bit weak and typing for extended DTDs is in some settings, e.g., streaming data, too complicated, it is natural to expect that a general-purpose XML schema language might have intermediate expressive power.

Two further classes of regular tree languages between DTD-definable and regular were proposed in [97]. To this end, two types are called *competing* if they are mapped by μ to the same element name. An extended DTD is **single-type** if none of its regular expressions contains two competing types. It is **restrained-competition** if there is no expression allowing two strings $u\sigma v$ and $u\tau w$ with competing types σ, τ.[8]

DTDs, single-type extended DTDs, restrained competition extended DTDs and unrestricted extended DTDs form a proper hierarchy.

It is argued in [97] that the W3C schema language XML schema is captured by single-type extended DTDs.

[7] [115] as well as some other papers use the term *specialized* instead of *extended*. Extended DTDs are implicit in [21] under the name *morphic images*.

[8] In [97] these classes are equivalently introduced in the framework of regular tree grammars.

Whereas it is clear that the above two restrictions entail efficient validation and typing, it is less obvious that they are also necessary for that purpose. Actually, one might suspect them to be a bit ad-hoc. Interestingly, it turns out that this is not at all the case. As shown in [85] both classes are quite robust and have a number of natural characterizations. Let the *ancestor-string* of a node be the sequence of labels on the path from the root to that node. For a regular tree language L the following are equivalent.

- L can be described by a single-type extended DTD.
- L has the following closure property, similar to the one of local tree languages mentioned above: if v, v' are nodes of trees $t, t' \in L$ with the same ancestor string. Then the tree resulting from replacing the subtree t_v of t rooted at v by $t'_{v'}$ is also in L.
- L can be described by a set of rules[9] $r \to s$, where r and s are regular expressions over the tree alphabet. Here, for different rules, the languages $L(r)$ have to be disjoint. A tree fulfills a rule $r \to s$ if for each node v with ancestor string in $L(r)$ the sequence of children labels is in $L(s)$. A tree fulfills a set of rules, if it fulfills each rule.

Some more characterizations are given in [85]. Further, languages definable by restrained-competition extended DTDs can be characterized similarly but with *ancestor-sibling strings* (giving the information about the ancestors of a node together with the left siblings of the ancestors) in place of ancestor strings. There are several consequences of these characterizations for the typing of streaming data which are discussed below. For DTD-definable languages a similar characterization by a closure property based only on node labels has been given in [115].

In [23] it was shown that each path closed regular tree language can be recognized by a deterministic tree-walking automaton. By combining the technique of that proof with the characterizations in [85] it can be shown that validity of trees with respect to single-type extended DTDs can be checked by deterministic tree-walking automata.

In [11] a closer inspection of the relationship between XML schema and the classes of the above hierarchy is done. Also an alternative specification mechanism is proposed based on the rules described above.

5.4. Complexity of validation and static analysis

5.4.1. Validation w.r.t. DTDs

Validation of a document t with respect to a (one-unambiguous) DTD d is algorithmically very simple, even if the DTD is not considered fixed but part of the input. It can be done in time $O(|t| \times |d|^2)$ by testing that for each non-leaf node the sequence of the labels of the children is accepted by the corresponding Glushkov automaton.

5.4.2. Static analysis of DTDs

Although one might be tempted to assume that for each DTD there exist valid documents, this is not true. As an example, let us consider the DTD with the single rule $a \to aa$ which only has an infinite model (the infinite binary tree consisting solely of as). Nevertheless, testing SATISFIABILITY of DTDs is a relatively easy task, possible in polynomial time. Further, by a straightforward reduction from PATHSYSTEMS [68], it can be shown that the problem is hard for P, even if each regular expression consists of a disjunction of strings with at most two symbols each.

Checking CONTAINMENT of DTDs can be much harder and the complexity depends strongly on the allowed regular expressions. In fact, in [84], it has been shown that, for every complexity class $\mathcal{C}$ containing P and obeying a simple closure property, the CONTAINMENT problem for DTDs with regular expressions from a class $\mathcal{R}$ is in $\mathcal{C}$ if and only if CONTAINMENT for regular expressions from $\mathcal{R}$ is in $\mathcal{C}$. In particular, for unrestricted regular expressions the problem is PSPACE-complete and for one-unambiguous expressions it is P-complete. In [84], the complexity of CONTAINMENT for many other kinds of simple regular expressions has been investigated. If the one-unambiguity requirement is dropped even very simple kinds of expressions (e.g., concatenations of symbols and expressions of the form $\sigma + \epsilon$) have an intractable CONTAINMENT problem.

[9] In [85], these rules were considered in the form of triples (r, σ, s), where σ was the label of the node under consideration.

5.4.3. Validation and static analysis w.r.t. extended DTDs

As a consequence of the close relationship between extended DTDs and non-deterministic parallel tree automata, the complexity of their basic algorithmic tasks coincides. E.g., VALIDATION is LOGCFL-complete (and can be done in time $O(|t| \times |\mathcal{A}|^2)$), SATISFIABILITY is complete for P and CONTAINMENT is complete for EXPTIME.

5.4.4. Static analysis for intermediate classes

The CONTAINMENT problem for the intermediate classes is generally easier than for extended DTDs. More precisely, CONTAINMENT for restrained-competition extended DTDs (as for DTDs) is PSPACE-complete but it is in PTIMEif the regular expressions are one-unambiguous [85].

5.5. Validating and typing streaming XML

In a scenario where XML data arrives in a streaming fashion, space and time efficiency of validation and typing become essential.

The validation of streaming XML data w.r.t. DTDs has been investigated in [123]. They show that validation is possible by a depth-synchronous pushdown automaton. It can be done even by a finite automaton if and only if the DTD is non-recursive, i.e., no element can occur inside an element with the same name, guaranteeing documents of bounded depth. A refined setting is to take as granted that documents are well-structured (i.e., opening and closing tags are nested correctly). In [123] a condition is presented that guarantees that a DTD can be validated by a finite automaton (under the well-formedness condition). It is conjectured that this condition is also sufficient and that the automaton construction given works in all cases. Validation for non-recursive one-unambiguous (and more generally, k-unambiguous) DTDs by finite document automata with an emphasis on the size of the resulting automata has been studied in [29].

Typing in a streaming context is more demanding. If types have to be assigned at the time an element is met (its opening or closing tag) decisions have to be taken without seeing the whole document. It turns out that the regular tree languages which allow this kind of typing have a surprisingly simple characterization. A regular tree language has an extended DTD that allows to assign a type to an element when its opening tag is met if and only if it is restrained competition [85]. In contrast, every regular tree language has an extended DTD which allows assignment of types at the closing tag. In both cases a deterministic depth-synchronous pushdown document automaton is sufficient. It basically simulates in a depth-first left-to-right manner a deterministic bottom–up automaton for the language.

The role of typing in stream query processing has been emphasized in [118]. They consider a two-phase automata-based model in which the first phase assigns types which are used in the second phase for query evaluation.

5.6. Related issues

Several papers studied which features of XML schema languages are used in practice. The structure of DTDs available from DTD repositories (XML.org) is analyzed in [30,119]. In contrast to frequent believe, recursion seems to be a frequently used feature of DTDs. More than half of the DTDs investigated in [30] used it. In [12] DTDs and XML schema descriptions available on the Web have been examined with a particular attention to the forms of regular expressions that are used.

The extension of schema languages by cardinality constraints has been proposed in several papers, e.g., [35,117]. They also consider extended automata models with (limited) counting abilities. Related work can be found in [125, 126]. For data-centric applications usually the order of elements is not important. To this end, the &-operator for "unordered concatenation" has been considered. Unfortunately, this operator is not well-suited for automata-based processing. E.g., the use of this operator might result in deterministic automata of exponential size [71]. The unambiguity requirement in the context of this operator is investigated in [98]. Another, value-based, extension of automata for the verification of key constraints can be found in [19].

A topic of its own is the maintenance of validity of documents under update. Automata-based approaches to this problem have been studied in [5,6]. A related issue is studied in [36]: how can a given document be slightly modified in order to be valid with respect to a given DTD or extended DTD? A grammar-based approach to this problem is proposed in [66]. A logic-based approach to constraint checking in an updating environment is proposed in [7]. In [39], the question is considered how schemata can be updated semi-automatically in order to keep updated documents valid.

In [131], deterministic string automata are used to implement efficient XML parsers in the context of Web services.

5.7. Section summary

DTDs have nice computational properties but limited expressive power in particular they lack types. Extended DTDs offer a robust framework for expressive schema languages, capturing the regular tree languages. There are important classes of intermediate expressive power which are computationally simpler than full extended DTDs, especially with respect to static analysis and in a streaming environment. The expressive power of XML schema is located in this intermediate level.

6. Navigation

As already mentioned in Section 2 the distinction between querying and navigation is not always easy. The main purpose of navigation is to identify a set of elements in a document. The specification of these elements might be relative to some given element or to a set S of elements of the document. Thus, in its most general form a navigational query specifies a binary query. Very often the specification is relative to the root element, in this case the query is unary or *node-selecting*. In Subsection 6.1, we first consider the class of *regular node-selecting queries* which, similar as the class of regular tree languages, offers a very robust framework for the study of node-selecting queries. In Subsection 6.2 we consider the main practical navigation language, XPath. Complexity results are surveyed in Subsection 6.3. After considering navigational queries on streaming XML data in 6.4, we turn to navigation viewed as binary queries in Subsection 6.5.

6.1. Regular node-selecting queries

It is natural to ask whether there is an analogous notion as regular tree languages for node-selecting queries. Actually, there is a very simple way to obtain such a generalization using a logical approach: simply consider queries that can be expressed by monadic second-order formulas with one free node variable. Each such formula defines on each tree a set of nodes. Whether this generalization can be considered reasonable depends on whether it leads to a robust class of queries, in particular, whether it has a counterpart based on parallel automata.

It turns out that this is actually the case. A first equivalent automata model can be obtained by equipping non-deterministic bottom–up automata with *selecting states*. A node is selected if the automaton visits it during an accepting run in a selecting state. As the automaton is non-deterministic it might of course happen that a node is selected in some accepting runs but not selected in some others. Nevertheless, one gets the same class of unary queries by either considering the nodes that are accepted during *at least one* accepting run or those accepted during *all* accepting runs. These automata have been called *selecting tree automata* in [48]. They exactly capture the class of unary queries defined by monadic second-order formulas with one free node variable [103].

An equivalent *deterministic* automaton model for unary queries has been proposed in [102]. It is easy to see that simple deterministic bottom–up automata are too weak to express all queries expressible by selecting tree automata. Therefore, following [24,93] *two-way automata* with the ability to move up and down in the tree along *cuts* were used. To get the power of selecting tree automata a further kind of transitions had to be added. They can be applied in situations where at each child of some node v there is a head of the automaton. Then a two-way string automaton with output is used to compute a new state assignment for these nodes. Deterministic two-way automata with selecting states and the ability to apply the latter kind of transitions at most once per node are exactly as expressive as selecting tree automata. They are called *query automata* in [102].

Although QUERYEVALUATION for tree automata is quite efficient, as will be discussed in Subsection 6.3 below, automata are not a good means to *specify* queries. For specification purposes, logical formulas are better suited but their QUERYEVALUATION complexity is much higher. *Monadic Datalog* [56] are very suitable as they allow to describe in a relatively simple way and QUERYEVALUATION is very efficient. In [56] even a very small fragment of Monadic Datalog is identified which can still express all regular node-selecting queries.

As there are many other equivalent models (e.g., [56,61,99–101,107]) defining the same class of unary queries it seems justified to call them the **regular node-selecting queries**.

In [27], regular node-selecting queries are expressed by *stepwise tree automata* using a particular encoding of unranked trees by binary trees. Yet another way of specifying regular node-selecting queries, based on regular hedge expressions is studied in [96].

A special kind of document automata is used in [99] to evaluate node-selecting queries. More precisely, they can be seen as depth-synchronous pushdown-automata which traverse the document twice, once in forward and once in backward direction. The first traversal computes additional (constant-size) information for each element, which is consumed during the second pass. A similar kind of two-pass processing has been proposed in [72] and can also be used for query automata and for the attributed grammars of [107].

6.2. XPath

Just like regular tree languages for XML schema languages, regular node-selecting queries constitute a fundamental framework for the study and classification of unary query formalisms. Nevertheless, in practical applications, XPath is the dominating language for navigation. In particular, it is an essential sublanguage for other languages as XQuery and XSLT but also for the specification of integrity constraints.[10]

As XPath is a practical language, it is not surprising that no exact correspondence with a clean automata class or logics is known. Nevertheless, in a sense it is more closely related to first-order logic than to MSO logic. More precisely, it is easy to see that the navigational core (called *Core XPath* in [56] and *Navigational XPath* in [90]), in which navigation along all axes and Boolean operations on filter expressions are allowed, can be captured by first-order logic. Actually, as shown in [87], with respect to unary queries, it exactly corresponds to two-variable logic. By adding the ability to repeat a certain navigation step arbitrarily (i.e., allowing expressions p^+, where p is a single navigation step possibly including a node test with a filter expression) one gets exactly the expressive power of first-order logic with respect to unary queries. The resulting query language is called *Conditional XPath*.[11] In [64] an even stronger fragment of XPath is considered, which allows also Boolean operations on the outer-most level. It actually extends Conditional XPath and can express queries as $(\texttt{child} :: *[F])^+$ by

$$\texttt{descendant} :: *[F] \cap \overline{\texttt{descendant} :: *[\mathit{not}\, F]/\texttt{descendant} :: *}.$$

As the automata we considered have the ability to count modulo some constant, there is no exact correspondence with first-order logic.[12] But it is reasonable to ask which kinds of automata can express all first-order or XPath-definable node-selecting queries.

In [15], it is not only shown that non-deterministic tree-walking automata do not capture the regular tree languages but also that there is a first-order definable language not expressible by an NTWA. Nevertheless, even deterministic pebble tree automata of the weakest kind mentioned in Section 4 (generalized to unranked trees) can evaluate all Boolean and unary first-order definable queries. This is not hard to see: assume the formula is in prenex normal form with k quantifiers. The TWA uses k pebbles which cycle through all possible assignments in a straightforward backtracking manner. As already mentioned, the strongest version of NTWAs captures exactly all Boolean, unary and binary queries definable in the extension of first-order logic by unary transitive closure. In particular, pebble automata can express all navigational XPath queries which will become especially important for type checking XSLT transformations as discussed in Section 7 below.

6.3. Complexity of navigational queries

6.3.1. Complexity of node-selecting queries

As already indicated above, the representation of a regular node-selecting query has a strong impact on the complexity of its evaluation. In general, logical formulas tend to have the highest complexity. More concretely, evaluation of a first-order formula on a tree is PSPACE-complete. Translating the formula into an automaton does not help as it might result in an automaton of *non-elementary*[13] size [47].

Evaluating selecting tree automata is possible in $O(|t||\mathcal{A}|^2)$ steps. This is implicit in [72]. Another advantage of Monadic Datalog, besides allowing relatively simple specification of queries is that the complexity of QUERYEVALUATION is $O(|t| \times |\mathcal{A}|)$.

[10] In this article, we only consider XPath 1.0 which has been the prime focus of theoretical investigations so far.

[11] [90] contains more sophisticated correspondences to logics, see also [75].

[12] As stated before, in [13] an automata-based characterization has been given.

[13] A function is *elementary* if it is bounded by a stack of exponentials of fixed size. Otherwise it is non-elementary.

How the size of a regular node-selecting query might change when moving from one representation (query language) to another is investigated in [61]. The comparison considers MSO logic, monadic Datalog, the modal μ-calculus with future and past modalities and several forms of monadic fixed point logics. MSO is by far the most succinct representation, followed by monadic fixed-point logic and its fragments.

Query evaluation on compressed XML data is studied in [48]. Documents are represented by directed acyclic graphs, where bisimilar nodes of the original tree get the same representative. If the query is given by a selecting tree automaton QUERYEVALUATION can be solved in exponential time in the size of the automaton and linear time in the compressed instance. For existential MSO-queries, QUERYEVALUATION becomes NEXPTIME-complete, arbitrary MSO-queries can be evaluated in exponential space.

The NON-EMPTINESS problem for node-selecting queries has basically the same complexity properties as for Boolean queries, i.e., it is in polynomial time. On the other hand, the CONTAINMENT problem becomes harder. It is EXPTIME-complete for non-deterministic automata but also, e.g., for query automata [102] and Monadic Datalog [56].

6.3.2. Complexity of XPath

There has been a lot of work on algorithmic and complexity theoretic aspects of XPath QUERYEVALUATION [53, 54] (see also the survey [57,58]) and on static analysis (SATISFIABILITY: [64], CONTAINMENT: [37,91,109,135], [121] is a survey), but only some of the methods use automata.

One nice feature of automata is their ability to be composed, therefore they can often be used in settings where several properties have to be checked at the same time. We consider here the example of the CONTAINMENT problem relative to a DTD for a fragment of XPath allowing expressions without filter predicates and only the child and descendant axis, and where every location step includes a node test. This problem is solvable in polynomial time as follows.[14] We are given a DTD d and two XPath expressions q_1, q_2 of the stated type and we have to check whether, for each tree t valid w.r.t. d, q_2 returns a superset of the result of q_1. This can be done by combining three non-deterministic top–down automata, one which checks validity with respect to d, one which checks that there exists a path fulfilling q_1 and one (actually deterministic one) which checks that there is no path fulfilling q_2.

It can also be shown by the use of tree automata that CONTAINMENT for the XPath fragment which additionally has filter expressions, wildcard and disjunction is in EXPTIME (it is actually EXPTIME-complete). This result is superseded by the EXPTIME upper bounds shown in [88,89] for a much larger fragment allowing navigation along all axes and even ID/IDREF constraints by reducing it to modal logics.

6.4. Node-selecting queries for streaming XML

The evaluation of node-selecting queries has also been studied in the context of streaming XML data and XML data in secondary storage. In [72] it is shown how regular node-selecting queries can be evaluated in two streaming passes basically by depth-synchronous pushdown document automata. The approach has some similarity with the one of [99].

Other work concentrates on the evaluation of XPath expressions, e.g., in a notification framework where many expressions are given, very many XML documents are read in a streaming fashion and it has to be quickly decided which queries match which documents. The use of deterministic string automata for this purpose is investigated in [60], pushdown automata are used in [62,116], and the cache performance of automata-based algorithms is studied in [63].

6.5. Navigation as binary queries

As stated before, in its most general form, navigation corresponds to binary queries, even though, e.g., XPath expressions are interpreted relative to the root node and thus define unary queries.

Every formalisms which describe a set of paths in a tree automatically induce a query with a binary relation result, namely the set of all pairs (u, v) of nodes having such a path from u to v. Again, MSO logic immediately induces a notion of regular binary queries.

[14] In [121] this result is claimed even in the case of wildcards, but this claim seems to be wrong.

How regular binary queries can be evaluated using two passes of a depth-synchronous pushdown automaton has been studied in [9]. A related approach can be found in [111]. We come back to this line of research in Section 8. The regular binary queries also exactly correspond to the binary relations (called *trips*) computable by a tree-walking automaton equipped with marbles and one pebble [43].

Another very natural way of specifying binary queries is by caterpillars, already discussed in Section 4. Nevertheless, in [23] they are only considered as means to define sets of nodes. Intuitively, a node v is contained in the result set if the caterpillar expression can be applied starting at v without getting stuck. In [52] caterpillar expressions are used to specify binary queries. By adding a limited form of negation and a mechanism to test whether a subexpression allows a path starting and ending at the current node, it is possible to express all first-order definable binary queries. It is also shown that such *looping caterpillars* have an automaton counterpart, namely pebble tree automata with a limited form of test whether the head is at the same node as a pebble (*non-inspecting pebble automata*). The regular binary queries can be obtained by adding regular node tests to looping caterpillars.

In [90] it is shown that Conditional XPath (cf. Subsection 6.1) exactly captures all first-order definable binary queries.

6.6. Section summary

Navigation is binary in general, but very often it boils down to unary, node-selecting, queries.

There is a robust class of node-selecting queries with the expressive power of MSO logic, equivalent automata models and many other characterizations. Evaluation is feasible from the data complexity point of view, CONTAINMENT is hard. XPath corresponds more to first-order logic and can be evaluated by seemingly weaker automata, e.g., pebble tree automata. There are many results on the complexity of QUERYEVALUATION and static analysis of various XPath fragments and there is also a solid body of work on the evaluation of XPath (and other node-selecting queries) in a streaming context.

There are several automata-based ways to capture the regular binary queries. XPath is basically captured by first-order logic and there is a natural extension of XPath which exactly corresponds to first-order logic. Caterpillar expressions are incomparable with first-order logic.

7. Transformations

Tree transformations is a core area of Formal Language Theory with many applications, e.g. in Programming Languages and Software Engineering. It involves quite complicated machinery and this survey can only scratch its surface.

General tree transformations are very powerful. It has been argued, e.g. in [10], that even a fragment of XSLT is more powerful than most XML query languages. This is mainly due to the fact that recursion is an inherent feature of general tree transformations whereas it is only an option (via user-definable functions) in XQuery.

In the following subsections we first survey top–down transducers, pebble transducers (most closely related to XSLT), macro tree transducers and some further transformation languages. Afterwards we turn to complexity issues.

7.1. Top–down transducers

The transformation language XSLT already has an "automata flavor" due to its usage of *modes* which are quite analogous to states of an automaton. Therefore, automata (in the context of transformations called *transducers*) are intensively studied as means to specify XML transformations.

Among the simplest transducers considered are *uniform top–down tree transducers* [82]. Such a transducer T has a set Q of states as usual tree automata and rules of the form $(p, \sigma) \to h$, where p is a state, σ is a symbol and h is a hedge (sequence of trees) with labels from an alphabet Σ and (only at leaves) from Q.

The semantics of T on an input tree t is best understood from the point of view of the output tree. An output tree t' is constructed in a top–down manner. At each stage, it has possibly some leaves labeled by states from Q. In the beginning, it has only one node, labeled by the initial state of the transducer. Furthermore, each such leaf v' corresponds to a node v of the input tree. The initial leaf (and root) of t' corresponds to the root of t. At a leaf v' of t' with state q a rule can be applied as follows. Let σ be the label of the corresponding node $v \in t$ and $u_1, \ldots, u_k$

its children. Then rules of the form $(q, \sigma) \to h$ can be applied. An application consists of replacing v' first by h and then replacing each leaf of h labeled with a state p by k new leaves labeled with p and corresponding to $u_1, \ldots, u_k$, respectively. Note that this process terminates at the leaves of t. These transducers are called *uniform* as every state at a leaf in h copies *all* children $u_1, \ldots, u_k$ of v rather than a selection of them. For examples of such transformations the reader is referred to [82].

A transducer might be non-deterministic, i.e., for the same pair (q, σ) there might exist several rules. Hence, in general, for each input tree t, $T(t)$ denotes the set of possible output trees.

XSLT can express such deterministic transformations but also much more expressive ones.

7.2. Pebble transducers

In [92,127], the core of XSLT (without consideration of data values) has been modeled by *pebble transducers*. They were defined for binary trees and used an encoding of XML documents as depicted in Fig. 1(e). The output tree is again produced from the root downwards. As before, some leaves of a partial output tree are labeled with states. But this time a leaf does not correspond only to a node of t but to a vector $(v_1, \ldots, v_l)$ of nodes of t, where $l \leqslant k$ for k-pebble transducers. Hence, a leaf has an associated configuration of a k-pebble automaton $\mathcal{A}$ on t. The computation of $\mathcal{A}$ might go on without producing output for some time but in some situations (depending on state, labels at pebbles and equality of nodes under pebbles) the computation might spawn two subcomputations which independently compute two output trees below the current leaf.

Whereas the output tree is constructed top–down as in the previous case, the access on the input tree is much more flexible and might involve going up and down. In fact, a uniform top–down tree transducer is basically a 0-pebble transducer which always moves downwards.

The latter can in turn be seen as attributed grammars (with trees as values), as discussed in Section 3.2 of [44]. For more background on the theory of top–down tree transducers, attribute grammars (as tree transducers), and macro tree transducers see the monograph [49].

7.3. Macro tree transducers

An even more involved transformation formalism is given by *macro tree transducers*. They also produce output in a top–down fashion and, like top–down transducers, they traverse the input tree top–down. But the current leaves in output trees may have parameters which are partial output trees themselves.

Interestingly, neither pebble transducers nor macro tree transducers are closed under composition but the class of transformations definable by compositions of pebble transducers and by compositions of macro tree transducers coincide. Actually, every pebble transducer has an equivalent composition of 0-pebble transducers, therefore compositions of 0-pebble transducers are equivalent to compositions of macro tree transducers [44].

An interesting special case is given by transformations of linear size increase, i.e., those for which the size of every output tree is linearly bounded in the size of the corresponding input tree. It is shown in [79] that every composition of deterministic macro tree transducers with this property can be effectively replaced by a single deterministic macro tree transducer.

7.4. Logically defined transducers

Tree transformations can also be defined by logical formulas in the way of logical interpretations. Actually, the result just mentioned at the end of the previous subsection is based on the fact that deterministic macro tree transducers with linear output size realize exactly the MSO-definable tree transformations, which are defined by logical formulas in the way of logical interpretations [45]. Further, this class is closed under composition, and that equivalence is decidable for transformations of this class [41].

A transformation language, DTL, closer to XSLT was proposed in [80]. It can be parametrized with different pattern matching facilities of which regular expressions and MSO-based pattern matching are studied. In the latter case the paper concentrates on the restriction where the input tree is traversed only in top–down fashion.

An extension of DTL with MSO pattern matching, unrestricted navigation in the input document, and accumulation of trees in parameters was introduced in [81] under the name TL. Whereas DTL-programs resemble automata quite

closely, TL has a more functional flavor (but this distinction is only gradual) and therefore more closely resembles XSLT. It is shown in [81] that for each deterministic TL-program there is an equivalent composition of three deterministic macro tree transducers, and for each non-deterministic TL-program there is an equivalent composition of three macro tree transducers (including one *stay macro tree transducer*). The corresponding translations are effective.

Another extension of DTL is proposed in [113,114].

7.5. Complexity of transformations

7.5.1. Evaluation

Although (composed) macro tree transducers are more powerful than, e.g., pebble tree transducers, they have an advantage with respect to the evaluation of transformations. Whereas pebble tree transducers might even be non-terminating (checking this for a given transducer is EXPTIME-complete), it is shown in [78] that (compositions of) deterministic macro tree transducers can be evaluated in time proportional to the sum of the sizes of the input tree plus the size of the output tree.

7.5.2. Static analysis

The most relevant static analysis problem for XML transformations is the TYPECHECKING problem, i.e., does a transformation T transform every input document valid for a schema d_1 into a document valid for a schema d_2. This problem is studied in settings where d_1 and d_2 are part of the input or fixed, respectively.

A tempting approach is via TYPEINFERENCE, i.e., by computing the set $T(d_1)$ of all possible output documents for input documents respecting d_1.[15] The drawback of this approach is that even for simple DTDs d_1 and transformations T, the set $T(d_1)$ might be non-regular. Interestingly, for the transformations studied here and regular tree languages, the reverse approach is more successful. In all cases the preimage of a regular set of trees under a transformation is again regular. This leads to the following approach to typechecking, first used in [92] in the case of pebble tree transducers: Compute a regular description d_2' of the documents *not* valid w.r.t. d_2. For this regular set compute a regular description d_1' of the preimage $T^{-1}(L(d_2'))$ and check that $L(d_1) \cap L(d_1') = \emptyset$.

This approach even works for the strongest kind of transformations mentioned above, compositions of macro tree transducers (or compositions of pebble automata) resulting in decidability of the TYPECHECKING problem for such transformations.

Unfortunately, the complexity in general is prohibitively high, non-elementary, i.e., not bounded by a fixed tower of exponentials. It can be shown that the complexity actually depends on the number k of pebbles for tree transducers or the number of macro tree transducers in a composition. The complexity is at most $(k+2)$-fold exponential for k-pebble transducers and at most $(k+1)$-fold exponential for compositions of k macro tree transducers.

In the light of this high complexity, in [82,83] the more tractable top–down tree transducers were studied. The complexities obtained in [82] range from P to EXPTIME where the polynomial time case is obtained when schemas are described by deterministic DTDs and transformations by non-deleting top–down transducers that are allowed to copy subtrees only in a bounded manner. In [83], a P-fragment is obtained for transformations that can delete as well as copy, but never at the same time. They correspond to so-called *filtering transformations* where interesting parts of the document are selected and others are deleted.

Extending transformations by the ability to refer to data values makes the TYPECHECKING problem undecidable but it remains decidable for restricted classes of transformations and schemas [3]. Typechecking of XML views of relational data is considered in [2].

7.5.3. Section summary

XML transformations are very powerful. The core capabilities (without considering data values) can be captured by different kinds of tree transducers and by other formal transformation languages. Deterministic transformations can be computed quite efficiently. The TYPECHECKING problem is generally decidable (again: no data values) but the complexity is quite high. Restricted cases with tractable typechecking have also been studied.

[15] The TYPEINFERENCE problem itself has been studied in [115].

8. Queries

Compared to the other XML processing tasks surveyed in this article, the foundations for general XML querying, e.g., by XQuery, are far less developed. This statement holds in particular with respect to automata-based query processing. It has to be investigated to which extent automata can be useful for general XML query processing.

We will only discuss two aspects of more general queries here, the construction of higher-arity results and the incorporation of data values and arithmetics.

Finally, we have a quick look at result concerning the evaluation of queries on streaming data.

8.1. Queries of higher arity

It should be noted first that, although the result of an XQuery formally is an XML document, XQuery clearly has relational and nested relational aspects, due to the bindings of variables to nodes and values.[16] Therefore, in order to better understand its expressive power, it is a useful step to consider (sub-)queries producing relational results of arbitrary arity.

Analogously to the case of unary and binary queries, there is a simple way to define queries of arity k by logical formulas with k free variables. There are also simple ways to define such queries by non-deterministic bottom–up tree automata. For instance, given a tree t and a tuple $x = (x_1, \ldots, x_k)$ of nodes of t one can define a tree $t[x]$ over an extended alphabet, in which the nodes x_i are labeled in a way indicating to which variables they are bound. For each MSO-definable query there is an automaton which accepts exactly those trees $t[x]$, for which x is in the query result. Although this approach can be useful for static analysis purposes, it is less attractive for query evaluation as each possible output tuple yields a different computation of the automaton.

Other approaches to expressing higher-arity queries are based on attribute grammars [107] (with an even super-MSO expressive power) and on tree grammars [99].

In [111] a particular class of queries is characterized allowing the evaluation of higher arity queries in linear time (in the size of input and output) by *unambiguous tree automata*.

Conjunctive queries, a weaker class of higher-arity queries is investigated in [59]. They are conjunctions of atomic path formulas based on the XPath axes and their closures. A very nice dichotomy based on the allowed axes holds for the combined complexity of QUERYEVALUATION. Conjunctive queries are tractable if they use (a subset of) one of the following sets of axes: {Child, NextSibling, NextSibling*, NextSibling$^+$}, {Child$^+$, Child*} or {Following}. For each other set of axes, the combined complexity of the QUERYEVALUATION problem is NP-complete.

Higher-arity queries based on logics and their decomposition have also been studied in [120].

8.2. Data values and arithmetics

Almost all investigations mentioned so far only consider the structure and the labels of XML documents but ignore the textual or data values. As discussed in Section 3 this approach might even be justified if queries refer to data values by comparisons with constant values. But it does not account for comparisons of data values at different nodes, hence if *joins* on data values are needed.

Automata (over words) that consume strings with data values have been investigated by several authors. Formally, instead of a finite alphabet one has an infinite set of data values and two data values can be compared for equality. In some investigations, each position of the input string carries a symbol from a finite alphabet and a data value.

Automata with registers are studied in [70]. In [20], restricted kinds of such automata were used in order to obtain a notion of regular languages. In [110], register-based and pebble-based automata were considered. For pebble automata, most models are equivalent (deterministic and non-deterministic, one-way and two-way) and the resulting class of languages lies between first-order and MSO-definable languages. Nevertheless, even NON-EMPTINESS for such automata is undecidable.

It is shown in [17] that, whereas first-order logic with three variables is undecidable for strings over data values, it is decidable for formulas with only two variables. Nevertheless the SATISFIABILITY problem has a high complexity,

[16] Of course, the data model is more complex, involving also lists and trees.

basically equivalent to reachability of Petri Nets. The decidability proof is by a reduction to *multicounter automata*. In [16], a similar result is shown for trees, in the presence of the `child` and `nextsibling` predicates.

The complexity of query evaluation for tree-walking automata which can store data values in registers has been investigated in [106]. Although these automata do not capture all first-order queries in general, on trees in which each node has a different value they are close to multi-head automata and capture complexity classes like LOGSPACE and, for suitable extensions, PTIME, PSPACE and EXPTIME.

The extension of (Boolean and unary) MSO-queries by some arithmetic abilities has been studied in [125,126]. There, transitions for automata (or the truth of logical formulas) can depend on linear equalities and inequalities over numbers of occurrences of states or true subformulas among the children of a node. Full MSO and non-deterministic tree automata extended by this feature quickly lead to undecidable problems in this setting [125]. Nevertheless, it is shown in [126] that adding them to deterministic automata or the modal μ-calculus does not increase the complexity of their basic static analysis problems considerably.

8.3. Queries on streaming XML

For general query processing, automata have mainly been used in the context of streaming XML. In [73] queries are specified in terms of attributed grammars and evaluated by deterministic depth-synchronous pushdown document automata. A stream-oriented language close to XQuery, FluX, is defined in [74]. It is shown how evaluation can be optimized in the presence of schema knowledge.

In [77] networks of *XML stream machines* are used to evaluate XQueries. In [67] automata are integrated into an query algebra. This seems to be a very interesting approach to use automata-based facilities for *subtasks* of query evaluation, as it seems unlikely that XQuery evaluation can be entirely based on automata.

8.3.1. Section summary

For general queries, in particular for XQuery evaluation, automata have not been much studied. Automata can be extended to capture higher arity queries and limited forms of counting and arithmetic. Automata models that are able to compare data values are usually undecidable. On streaming XML, (document) automata have been employed successfully.

9. Conclusion

Automata and formal language theory are important for the foundations of XML processing. In particular, they offer a robust framework of *regularity* for tree languages, node-selecting and binary queries. Nevertheless, the expressive power of practical languages usually does not fully cover these regular classes. For XML transformations, tree transducers are a useful framework. In the context of schemas, navigation and transformation, automata have been used as algorithmic models but also as tools in proofs and to facilitate static analysis. The question whether automata will be useful also for general querying has not been settled yet.

Acknowledgments

The author warmly thanks Joost Engelfriet, Christoph Koch, Sebastian Maneth, Wim Martens, Maarten Marx, Frank Neven, and Volker Weber and two anonymous for very many valuable suggestions.

References

[1] Alfred V. Aho, Jeffrey D. Ullman, Translations on a context-free grammar, Inform. Control 19 (5) (1971) 439–475.
[2] Noga Alon, Tova Milo, Frank Neven, Dan Suciu, Victor Vianu, Typechecking XML views of relational databases, ACM Trans. Comput. Logic 4 (3) (2003) 315–354.
[3] Noga Alon, Tova Milo, Frank Neven, Dan Suciu, Victor Vianu, XML with data values: Typechecking revisited, J. Comput. System Sci. 66 (4) (2003) 688–727.
[4] Rajeev Alur, P. Madhusudan, Visibly pushdown languages, in: STOC 2004, pp. 202–211.
[5] Andrey Balmin, Yannis Papakonstantinou, Victor Vianu, Incremental validation of XML documents, ACM Trans. Database Syst. 29 (4) (2004) 710–751.

[6] Denilson Barbosa, Alberto O. Mendelzon, Leonid Libkin, Laurent Mignet, Marcelo Arenas, Efficient incremental validation of XML documents, in: ICDE 2004, 2004, pp. 671–682.
[7] Michael Benedikt, Glenn Bruns, Julie Gibson, Robin Kuss, Amy Ng, Automated update management for XML integrity constraints, Bell Labs, Lucent Technologies, 2002.
[8] Michael Benedikt, Luc Segoufin, Regular tree languages definable in FO, in: STACS 2005, 2005, pp. 327–339.
[9] Alexandru Berlea, Helmut Seidl, Binary queries, in: Extreme Markup Languages 2002, 2002.
[10] Geert Jan Bex, Sebastian Maneth, Frank Neven, A formal model for an expressive fragment of XSLT, Inform. Syst. 27 (1) (2002) 21–39.
[11] Geert Jan Bex, Wim Martens, Frank Neven, Thomas Schwentick, Expressiveness of XSDs: From practice to theory, there and back again, in: WWW 2005, 2005.
[12] Geert Jan Bex, Frank Neven, Jan Van den Bussche, DTDs versus XML schema: A practical study, in: WebDB 2004, 2004, pp. 79–84.
[13] Mikolaj Bojanczyk, Decidable properties of tree languages, PhD thesis, Warsaw University, 2004.
[14] Mikolaj Bojanczyk, Thomas Colcombet, Tree-walking automata cannot be determinized, in: Proc. 31st International Colloq. on Automata, Languages, and Programming (ICALP), Turku, 2004, pp. 246–256.
[15] Mikolaj Bojanczyk, Thomas Colcombet, A regular tree language not recognized by any tree-walking automaton, in: STOC 2005, 2005.
[16] Mikolaj Bojanczyk, Claire David, Anca Muscholl, Thomas Schwentick, Luc Segoufin, Two-variable logics on data trees and XML reasoning, 2005.
[17] Mikolaj Bojanczyk, Claire David, Anca Muscholl, Thomas Schwentick, Luc Segoufin, Two-variable logics on words with data, 2006.
[18] A. Bonifati, D. Lee, Technical survey of XML schema and query languages, 2000.
[19] Beatrice Bouchou, Mirian Halfeld Ferrari Alves, Martin A. Musicante, Tree automata to verify XML key constraints, in: WebDB 2003, 2003, pp. 37–42.
[20] Patricia Bouyer, Antoine Petit, Denis Therien, An algebraic approach to data languages and timed languages, Inform. and Comput. 182 (2) (2003) 137–162.
[21] A. Brüggemann-Klein, M. Murata, D. Wood, Regular tree languages over non-ranked alphabets, available from http://hdl.handle.net/1783.1/738, 1998.
[22] A. Brüggemann-Klein, D. Wood, One-unambiguous regular languages, Inform. and Comput. 140 (2) (1998) 229–253.
[23] Anne Brüggemann-Klein, Derick Wood, Caterpillars: A context specification technique, Markup Languages 2 (1) (2000) 81–106.
[24] Anne Brüggemann-Klein, Derick Wood, The regularity of two-way nondeterministic tree automata languages, Int. J. Found. Comput. Sci. 13 (1) (2002) 67–81.
[25] Diego Calvanese, Giuseppe De Giacomo, Maurizio Lenzerini, Moshe Y. Vardi, Reasoning on regular path queries, SIGMOD Record 32 (4) (2003) 83–92.
[26] Diego Calvanese, Giuseppe De Giacomo, Maurizio Lenzerini, Moshe Y. Vardi, View-based query containment, in: PODS 2003, 2003, pp. 56–67.
[27] Julien Carme, Joachim Niehren, Marc Tommasi, Querying unranked trees with stepwise tree automata, in: RTA 2004, 2004.
[28] Boris Chidlovskii, Using regular tree automata as XML schemas, in: ADL 2000, 2000, pp. 89–104.
[29] Cristiana Chitic, Daniela Rosu, On validation of XML streams using finite state machines, in: WebDB 2004, 2004, pp. 85–90.
[30] Byron Choi, What are real DTDs like? in: WebDB 2002, 2002, pp. 43–48.
[31] J. Clark, M. Makoto, Relax NG specification, Technical report, OASIS Committee Specification Document, 2001.
[32] H. Comon, M. Dauchet, R. Gilleron, F. Jacquemard, D. Lugiez, S. Tison, M. Tommasi, Tree automata techniques and applications, available on: http://www.grappa.univ-lille3.fr/tata, 1997, release October 1st, 2002.
[33] Bruno Courcelle, On recognizable sets and tree automata, in: Maurice Nivat, Hassan Ait-Kaci (Eds.), Resolution of Equations in Algebraic Structures, Academic Press, 1989.
[34] Julien Cristau, Christof Löding, Wolfgang Thomas, Deterministic automata on unranked trees, 2005.
[35] Silvano Dal-Zilio, Denis Lugiez, XML schema, tree logic and sheaves automata, in: RTA 2003, 2003, pp. 246–263.
[36] Michel de Rougemont, The correction of XML data, in: First Franco-Japanese Workshop on Information Search, Integration and Personalization, ISIP '03, 2003.
[37] Alin Deutsch, Val Tannen, Containment and integrity constraints for XPath fragments, in: KRDB '01, 2001.
[38] John Doner, Tree acceptors and some of their applications, J. Comput. System Sci. 4 (5) (1970) 406–451.
[39] D. Duarte, B. Bouchou, M. Halfeld Ferrari Alves, D. Laurent, Incremental schema update for XML documents, 2002.
[40] J. Engelfriet, H.J. Hoogeboom, Automata with nested pebbles capture first-order logic with transitive closure, Technical report 2005-02, LIACS, April 2005.
[41] J. Engelfriet, S. Maneth, The equivalence problem for deterministic MSO tree transducers is decidable, 2005.
[42] Joost Engelfriet, Hendrik Jan Hoogeboom, Tree-walking pebble automata, in: Jewels are Forever, Contributions on Theoretical Computer Science in Honor of Arto Salomaa, Springer, London, UK, 1999, pp. 72–83.
[43] Joost Engelfriet, Hendrik Jan Hoogeboom, Jan-Pascal Van Best, Trips on trees, Acta Cybernet. 14 (1) (1999) 51–64.
[44] Joost Engelfriet, Sebastian Maneth, A comparison of pebble tree transducers with macro tree transducers, Acta Inform. 39 (9) (2003) 613–698.
[45] Joost Engelfriet, Sebastian Maneth, Macro tree translations of linear size are MSO definable, SIAM J. Comput. 32 (1) (2003) 950–1006.
[46] Joost Engelfriet, Grzegorz Rozenberg, Giora Slutzki, Tree transducers, L systems, and two-way machines, J. Comput. System Sci. 20 (1980) 150–202.
[47] Markus Frick, Martin Grohe, The complexity of first-order and monadic second-order logic revisited, in: LICS 2002, 2002, pp. 215–224.
[48] Markus Frick, Martin Grohe, Christoph Koch, Query evaluation on compressed trees (extended abstract), in: LICS 2003, 2003, pp. 188–197.
[49] Zoltán Fülöp, Heiko Vogler, Syntax-Directed Semantics—Formal Models Based on Tree Transducers, Monogr. Theoret. Comput. Sci. EATCS Ser., Springer, 1998.

[50] F. Gecseg, M. Steinby, Tree Automata, Akademiai Kiado, Budapest, 1984.
[51] V.M. Glushkov, The abstract theory of automata, Russian Math. Surveys 16 (1961) 1–53.
[52] Evan Goris, Maarten Marx, Looping caterpillars, in: LICS 2005, 2005.
[53] G. Gottlob, C. Koch, R. Pichler, Efficient algorithms for processing XPath queries, in: Proc. 28th International Conference on Very Large Data Bases (VLDB), Hongkong, 2002, pp. 95–106.
[54] G. Gottlob, C. Koch, R. Pichler, The complexity of XPath query evaluation, in: Proc. 22nd Symposium on Principles of Database Systems (PODS), San Diego, 2003.
[55] Georg Gottlob, Christoph Koch, Monadic datalog and the expressive power of languages for web information extraction, in: PODS 2002, 2002, pp. 17–28.
[56] Georg Gottlob, Christoph Koch, Monadic datalog and the expressive power of languages for web information extraction, J. ACM 51 (1) (2004) 74–113.
[57] Georg Gottlob, Christoph Koch, Reinhard Pichler, XPath processing in a nutshell, SIGMOD Record 32 (2) (2003) 21–27.
[58] Georg Gottlob, Christoph Koch, Reinhard Pichler, XPath processing in a nutshell, SIGMOD Record 32 (1) (2003) 12–19.
[59] Georg Gottlob, Christoph Koch, Klaus U. Schulz, Conjunctive queries over trees, in: PODS 2004, 2004, pp. 189–200.
[60] Todd J. Green, Gerome Miklau, Makoto Onizuka, Dan Suciu, Processing XML streams with deterministic automata, in: ICDT 2003, 2003, pp. 173–189.
[61] Martin Grohe, Nicole Schweikardt, Comparing the succinctness of monadic query languages over finite trees, in: CSL 2003, 2003, pp. 226–240.
[62] Ashish Kumar Gupta, Dan Suciu, Stream processing of XPath queries with predicates, in: SIGMOD Conference 2003, 2003, pp. 419–430.
[63] Bingsheng He, Qiong Luo, Byron Choi, Cache-conscious automata for XML filtering, in: ICDE 2005, 2005.
[64] Jan Hidders, Satisfiability of XPath expressions, in: DBPL 2003, 2003, pp. 21–36.
[65] Haruo Hosoya, Benjamin C. Pierce, Regular expression pattern matching for XML, J. Funct. Program. 13 (6) (2003) 961–1004.
[66] Ionut Emil Iacob, Alex Dekhtyar, Michael I. Dekhtyar, Checking potential validity of XML documents, in: WebDB 2004, 2004, pp. 91–96.
[67] Jinhui Jian, Hong Su, Elke A. Rundensteiner, Automaton meets query algebra: Towards a unified model for XQuery evaluation over XML data streams, in: ER 2003, 2003, pp. 172–185.
[68] Neil D. Jones, William T. Laaser, Complete problems for deterministic polynomial time, in: STOC '74: Proceedings of the Sixth Annual ACM Symposium on Theory of Computing, ACM Press, New York, NY, 1974, pp. 40–46.
[69] Eija Jurvanen, Andreas Potthoff, Wolfgang Thomas, Tree languages recognizable by regular frontier check, in: Developments in Language Theory 1993, 1993, pp. 3–17.
[70] Michael Kaminski, Nissim Francez, Finite-memory automata, Theoret. Comput. Sci. 134 (2) (1994) 329–363.
[71] Pekka Kilpeläinen, SGML & XML content models, Markup Languages 1 (2) (1999) 53–76.
[72] Christoph Koch, Efficient processing of expressive node-selecting queries on XML data in secondary storage: A tree automata-based approach, in: VLDB 2003, 2003, pp. 249–260.
[73] Christoph Koch, Stefanie Scherzinger, Attribute grammars for scalable query processing on XML streams, in: DBPL 2003, 2003, pp. 233–256.
[74] Christoph Koch, Stefanie Scherzinger, Nicole Schweikardt, Bernhard Stegmaier, Schema-based scheduling of event processors and buffer minimization for queries on structured data streams, in: VLDB 2004, 2004, pp. 228–239.
[75] Leonid Libkin, Logics for unranked trees: An overview, in: Proceedings of ICALP 2005, 2005, pp. 35–50.
[76] Markus Lohrey, On the parallel complexity of tree automata, in: RTA 2001, 2001, pp. 201–215.
[77] Bertram Ludäscher, Pratik Mukhopadhyay, Yannis Papakonstantinou, A transducer-based XML query processor, in: VLDB 2002, 2002, pp. 227–238.
[78] Sebastian Maneth, The complexity of compositions of deterministic tree transducers, in: FSTTCS 2002, 2002, pp. 265–276.
[79] Sebastian Maneth, The macro tree transducer hierarchy collapses for functions of linear size increase, in: FSTTCS 2003, 2003, pp. 326–337.
[80] Sebastian Maneth, Frank Neven, Structured document transformations based on XSL, in: DBPL 1999, 1999, pp. 80–98.
[81] Sebastian Maneth, Thomas Perst, Alexandru Berlea, Helmut Seidl, XML type checking with macro tree transducers, in: PODS 2005, 2005.
[82] Wim Martens, Frank Neven, Typechecking top–down uniform unranked tree transducers, in: ICDT '03, 2003.
[83] Wim Martens, Frank Neven, Frontiers of tractability for typechecking simple XML transformations, in: PODS 2004, 2004, pp. 23–34.
[84] Wim Martens, Frank Neven, Thomas Schwentick, Complexity of decision problems for simple regular expressions, in: MFCS 2004, 2004, pp. 889–900.
[85] Wim Martens, Frank Neven, Thomas Schwentick, Which XML schemas admit 1-pass preorder typing? in: ICDT 2005, 2005, pp. 68–82.
[86] Wim Martens, Joachim Niehren, Minimizing tree automata for unranked trees, in: DBPL, 2005, pp. 232–246.
[87] M. Marx, M. de Rijke, Semantic characterizations of XPath, in: TDM '04 workshop on XML Databases and Information Retrieval, Twente, The Netherlands, June 21, 2004.
[88] Maarten Marx, XPath and modal logics of finite DAG's, in: TABLEAUX 2003, 2003, pp. 150–164.
[89] Maarten Marx, XPath with conditional axis relations, in: EDBT 2004, 2004.
[90] Maarten Marx, First order paths in ordered trees, in: Proc. 10th Int. Conf. on Database Theory (ICDT), Edinburgh, 2005, pp. 114–128.
[91] Gerome Miklau, Dan Suciu, Containment and equivalence for a fragment of XPath, J. ACM 51 (1) (2004) 2–45.
[92] Tova Milo, Dan Suciu, Victor Vianu, Typechecking for XML transformers, J. Comput. System Sci. 66 (1) (2003) 66–97.
[93] Etsuro Moriya, On two-way tree automata, Inform. Process. Lett. 50 (3) (1994) 117–121.
[94] Makoto Murata, Forest-regular languages and tree-regular languages, Technical report, Fuji Xerox, Japan, 1995.
[95] Makoto Murata, Hedge automata: A formal model for XML schemata, Fuji Xerox Information Systems, http://www.oasis-open.org/cover/hedge20000224.html, 2000.

[96] Makoto Murata, Extended path expressions for XML, in: PODS 2001, 2001.
[97] Makoto Murata, Dongwon Lee, Murali Mani, Taxonomy of XML schema languages using formal language theory, in: Extreme Markup Languages, Montreal, Canada, August 2001, 2001.
[98] Andreas Neumann, Unambiguity of SGML content models—Pushdown automata revisited, in: Universität Trier, Mathematik/Informatik, Forschungsbericht 97-05, 1997.
[99] Andreas Neumann, Parsing and querying XML documents in SML, PhD thesis, University of Trier, 1999.
[100] Andreas Neumann, Helmut Seidl, Locating matches of tree patterns in forests, in: FSTTCS 1998, 1998, pp. 134–145.
[101] F. Neven, Attribute grammars for unranked trees as a query language for structured documents, J. Comput. System Sci. 70 (2005) 221–257.
[102] F. Neven, T. Schwentick, Query automata on finite trees, Theoret. Comput. Sci. 275 (2002) 633–674.
[103] Frank Neven, Design and analysis of query languages for structured documents—A formal and logical approach, PhD thesis, Limburgs Universitair Centrum, 1999.
[104] Frank Neven, Automata, logic, and XML, in: Computer Science Logic, 16th International Workshop, (CSL), 2002, pp. 2–26.
[105] Frank Neven, Automata theory for XML researchers, SIGMOD Record 31 (3) (2002).
[106] Frank Neven, On the power of walking for querying tree-structured data, in: PODS 2002, 2002, pp. 77–84.
[107] Frank Neven, Jan Van den Bussche, Expressiveness of structured document query languages based on attribute grammars, J. ACM 49 (1) (2002) 56–100.
[108] Frank Neven, Thomas Schwentick, On the power of tree-walking automata, Inform. and Comput. 183 (1) (2003) 86–103.
[109] Frank Neven, Thomas Schwentick, XPath containment in the presence of disjunction, DTDs, and variables, in: ICDT 2003, 2003, pp. 315–329.
[110] Frank Neven, Thomas Schwentick, Victor Vianu, Finite state machines for strings over infinite alphabets, ACM Trans. Comput. Logic 15 (3) (2004) 403–435.
[111] Joachim Niehren, Laurent Planque, Jean-Marie Talbot, Sophie Tison, N-ary queries by tree automata, 2004.
[112] Maurice Nivat, Andreas Podelski, Minimal ascending and descending tree automata, SIAM J. Comput. 26 (1) (1997) 39–58.
[113] Tadeusz Pankowski, Transformation of XML data using an unranked tree transducer, in: EC-Web 2003, 2003, pp. 259–269.
[114] Tadeusz Pankowski, A high-level language for specifying XML data transformations, in: ADBIS 2004, 2004, pp. 159–172.
[115] Yannis Papakonstantinou, Victor Vianu, DTD inference for views of XML data, in: PODS 2000, ACM, 2000, pp. 35–46.
[116] Feng Peng, Sudarshan S. Chawathe, XPath queries on streaming data, in: SIGMOD Conference 2003, 2003, pp. 431–442.
[117] Florian Reuter, Norbert Luttenberger, Cardinality constraint automata: A core technology for efficient XML schema-aware parsers, 2003.
[118] George Russell, Mathias Neumüller, Richard C.H. Connor, Typex: A type based approach to XML stream querying, in: WebDB 2003, 2003, pp. 55–60.
[119] Arnaud Sahuguet, Everything you ever wanted to know about DTDs, but were afraid to ask, in: WebDB (Informal Proceedings) 2000, 2000, pp. 69–74.
[120] Thomas Schwentick, On diving in trees, in: MFCS 2000, 2000, pp. 660–669.
[121] Thomas Schwentick, XPath query containment, SIGMOD Record 33 (1) (2004) 101–109.
[122] Luc Segoufin, Typing and querying XML documents: Some complexity bounds, in: PODS 2003, 2003, pp. 167–178.
[123] Luc Segoufin, Victor Vianu, Validating streaming XML documents, in: PODS 2002, 2002, pp. 53–64.
[124] Helmut Seidl, Deciding equivalence of finite tree automata, SIAM J. Comput. 19 (3) (1990) 424–437.
[125] Helmut Seidl, Thomas Schwentick, Anca Muscholl, Numerical document queries, in: Proc. 22nd Symposium on Principles of Database Systems (PODS), San Diego, 2003, pp. 155–166.
[126] Helmut Seidl, Thomas Schwentick, Anca Muscholl, Peter Habermehl, Counting in trees for free, in: ICALP 2004, 2004, pp. 1136–1149.
[127] Dan Suciu, Typechecking for semistructured data, in: DBPL 2001, 2001, pp. 1–20.
[128] James W. Thatcher, Jesse B. Wright, Generalized finite automata theory with an application to a decision problem of second-order logic, Math. Syst. Theory 2 (1) (1968) 57–81.
[129] W. Thomas, Languages, automata, and logic, in: G. Rozenberg, A. Salomaa (Eds.), Handbook of Formal Languages, vol. III, Springer, 1997, pp. 389–455.
[130] Giora Slutzki Tsutomu Kamimura, Parallel and two-way automata on directed ordered acyclic graphs, Inform. Control 49 (1981) 10–51.
[131] Robert van Engelen, Constructing finite state automata for high-performance XML web services, in: International Conference on Internet Computing 2004, 2004, pp. 975–981.
[132] Moshe Y. Vardi, Logic and automata: A match made in heaven, in: ICALP 2003, 2003, pp. 64–65.
[133] M. Veanes, On computational complexity of basic decision problems of finite tree automata, Technical report, UPMAIL Technical Report 133, Uppsala University, Computing Science Department, 1997.
[134] Victor Vianu, A Web odyssey: From Codd to XML, SIGMOD Record 32 (2) (2003) 68–77.
[135] Peter T. Wood, Containment for XPath fragments under DTD constraints, in: ICDT 2003, 2003, pp. 300–314.
[136] François Yergeau, Tim Bray, Jean Paoli, C.M. Sperberg-McQueen, Eve Maler, Extensible markup language (XML) 1.0, third ed., W3C recommendation, February 2004.