XML: From practice to theory

Victor Vianu*

U.C. San Diego
La Jolla, CA 92093-0114, USA
vianu@cs.ucsd.edu

Abstract

The development of XML technology has occurred very rapidly, initially leaving theory behind. As is often the case in such situations, practical development sometimes seemed more ad-hoc than well principled. There is now a substantial body of work providing formal foundations for XML. This paper presents a personal perspective on some recent developments in the theory of XML.

1. Introduction

As the standard for data exchange on the Web, XML has become a central component of modern data management. Unlike the relational model, whose birth was preceded by solid theoretical foundations, XML is the fruit of an often chaotic design process involving many different communities and standards committees. While the theory of XML initially lagged behind the practice, this is beginning to change. There is now a substantial body of work providing formal foundations for XML, including schemas, constraints, and query languages. In this paper we describe some of the recent developments in the theory of XML, emphasizing a very useful and elegant connection of XML schemas and queries to tree automata and transducers. Indeed, this connection has emerged as a recurring theme in recent theoretical research on XML.

We begin with a brief review of XML, then discuss schema formalisms and their connection to tree automata theory. We next proceed to XML query languages and their abstraction as tree transducers. Next, we show how this can be applied to static typechecking. Finally, we present an informal overview of other research on XML, including constraints for XML, the complexity of XPath query evaluation and containment testing, validation of streaming XML, incremental XML validation, and XML with incomplete information.

This paper is not meant to be a comprehensive survey of the developments in database theory related to XML. Several excellent survey articles are referenced in the paper. The book [1] is a valuable source of information on databases and the Web, including semi-structured data and XML.

2. XML Documents and Schemas

XML documents as labeled trees XML has been introduced in the document community as a subset of SGML. XML is in some sense an augmentation of HTML allowing annotations of data with information about its *meaning* rather than just its presentation. An XML document consists of nested elements, with ordered sub-elements. Each element has a name (also called tag or label). The full XML has many bells and whistles, but its simplest abstraction is as a

*Work supported in part by the NSF under grants ITR-0225676 (SEEK) and ITR-0313384.

```
<dealer>
 <UsedCars>
  <ad>
   <model>Honda</model>
   <year>92</year>
  </ad>
 </UsedCars>
 <NewCars>
  <ad>
   <model>BMW</model>
  </ad>
 </NewCars>
</dealer>
```

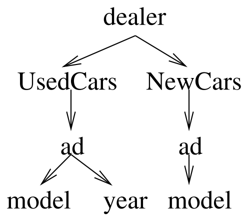


Figure 1: Dealer XML document

labeled ordered tree (with labels on nodes), possibly with data values associated to the leaves. For example, an XML document holding ads for used cars and new cars is shown in Figure 1 (left), together with its abstraction as a labeled tree (right, data values omitted).

The emergence of XML has placed increased importance on labeled trees capturing the structure of XML documents. However, XML additionally provides a referencing mechanism among elements that allows simulating arbitrary graphs, and so, semi-structured data. This aspect has been left out of some formal models for XML, because XML schemas and query languages typically do not take advantage of it.

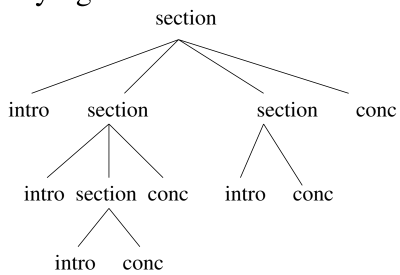
It is worth mentioning that XML can also be viewed as an object model. This is illustrated by a standard API for XML proposed by W3C, where XML documents are described in terms of the *Document Object Model* (DOM). Other extensions to XML and DTDs proposed by W3C, such as RDF, also have an object-oriented flavor (see [1] and the W3C Web site).

XML schemas The basic schema mechanism for XML is provided by Data Type Definitions (DTDs). More recently, many schema languages extending DTDs have been proposed, including XML Schema, DSD, SOX, RELAX, etc. Most proposals can be found as technical reports of the W3C (http://www.w3c.org). Comparative presentations of several XML schema languages can be found in [1] and [40]. A survey of the XML schema languages is beyond the scope of this paper. We will focus here on DTDs and a very useful extension present in XML Schema, allowing to decouple the definition of the structure of an element from its name.

Essentially, a DTD is an extended context-free grammar. The non-terminals of the grammar are the labels (tags) of elements in the labeled tree corresponding to the XML document. There are no terminal symbols. Let Σ be a finite alphabet of labels. A DTD consists of a set of rules of the form $e \to r$ where $e \in \Sigma$ and r is a regular expression over Σ. There is one such rule for each e, and the DTD also specifies the label of the root. An XML document satisfies a DTD if it is a derivation of the extended context-free grammar. That is, for label e with associated rule $e \to r$, and each node labeled e, the sequence of labels of its children spells a word in r. For example, a DTD might consist of the rules (with ϵ-rules omitted):

$$root : section;$$
$$section \to intro, section^*, conc$$

An example of a labeled tree satisfying the above DTD is:



Thus, each DTD d defines a set of labeled ordered trees, denoted by $sat(d)$.

It turns out that DTDs have many limitations as schema languages. Some are addressed in XML Schema and in other extensions that have been proposed. For example, the set of tree languages definable by DTDs is not even closed under union (nor other Boolean operators). Another important limitation is the inability to separate the *type* of an element from its *name*. For example, consider the dealer document in Figure 1. A DTD corresponding to it might consist of the rules:

root: *dealer*	*dealer* $\rightarrow$ *UsedCars NewCars*	*ad* $\rightarrow$ *model year* $\mid$ *model*
UsedCars $\rightarrow$ *ad**	*NewCars* $\rightarrow$ *ad**	

However, it may be natural for used car ads to have different structure than new car ads. There is no mechanism to do this using DTDs, since rules depend only on the name of the element, and not on its context. To overcome this limitation, extensions of DTDs provide mechanisms to decouple element names from their types and thus allow context-dependent definitions of their structure. Interestingly, this also leads to closure of the definable sets of trees under Boolean operations. We show one way to formalize the decoupling of names from types, using the notion of *specialized DTD* (studied in [56] and similar to formalisms proposed in [13, 21] and adopted in restricted form in XML Schema and XQuery). The idea is to use whenever necessary "specializations" of element names with their own type definition. More precisely, a *specialized DTD* for alphabet Σ is a 4-tuple $\langle \Sigma, \Sigma', d, \mu \rangle$ where:

1. Σ, Σ' are finite alphabets;
2. d is an DTD over Σ'; and,
3. μ is a mapping from Σ' to Σ.

Intuitively, Σ' provides for some $a \in \Sigma$, a set of specializations of a, namely those $a' \in \Sigma'$ for which $\mu(a') = a$. Note that μ induces a homomorphism on words over Σ', and also on trees over Σ' (yielding trees over Σ). We also denote by μ the induced homomorphisms. Let us denote specialized DTDs by bold letters **d,e,f**, etc.

Let **d** = $\langle \Sigma, \Sigma', d, \mu \rangle$ be a specialized DTD. A tree t over Σ satisfies **d** if $t \in \mu(sat(d))$. Thus, t is a homomorphic image under μ of a derivation tree in d. Equivalently, a labeled tree over Σ is valid if it can be specialized to a tree which is valid with respect to the DTD over the specialized alphabet.

For example, we can now write a specialized DTD distinguishing used car ads from new car ads in the dealer example as follows. $\Sigma = \{$*dealer, UsedCars, NewCars, ad, model, year*$\}$, $\Sigma' = \Sigma \cup \{ad^{used}, ad^{new}\}$, μ is the identity on Σ and $\mu(ad^{used}) = \mu(ad^{new}) = ad$ and the DTD over Σ' (with ϵ-rules omitted) is:

root: *dealer*	*dealer* $\rightarrow$ *UsedCars NewCars*	*UsedCars* $\rightarrow (ad^{used})^*$
NewCars $\rightarrow (ad^{new})^*$	$ad^{used} \rightarrow$ *model year*	$ad^{new} \rightarrow$ *model*

Many interesting basic questions arise in connection to XML schemas. How hard is it to check validity of an XML document with respect to a schema? When can a set of XML documents be characterized by a schema? Is there always a most precise schema describing a given set of XML documents? Can the union, difference, intersection of sets of valid documents specified by schemas be in turn described by another schema? If yes, how can that schema be computed? We next discuss a powerful, effective tool for dealing with such questions: a remarkable connection between schemas and tree automata.

XML schemas, tree automata, and logic. We informally review the notion of regular tree language and tree automaton. Tree automata are devices whose function is to accept or reject their input, which in the classical framework is a complete binary tree with nodes labeled with symbols from some finite alphabet Σ. There are several equivalent variations of tree automata. A non-deterministic top-down tree automaton over Σ has a finite set Q of states, including a distinguished initial state q_0 and an accepting state q_f. In a computation, the automaton labels the nodes of the tree with states, according to a set of rules, called *transitions*. An internal node transition is of the form $(a, q) \to (q', q'')$, for $a \in \Sigma$. It says that, if an internal node has symbol a and is labeled by state q, then its left and right children may be labeled by q' and q'', respectively. A leaf transition is of the form $(a, q) \to q_f$ for $a \in \Sigma$. It allows changing the label of a leaf with symbol a from q to the accepting state q_f. Each computation starts by labeling the root with the start state q_0, and proceeds by labeling the nodes of the trees non-deterministically according to the transitions. The input tree is accepted if *some* computation results in labeling all leaves by q_f. A set of complete binary trees is *regular* iff it is accepted by some top-down tree automaton. Deterministic and non-deterministic bottom-up automata can also be defined, and they are both equivalent to the non-deterministic top-down automata.

There is a strong connection between regular tree languages and logic, similar to the string case. Regular tree languages are precisely those definable by Monadic Second-Order logic (MSO) on structures representing binary trees. Regular languages of finite binary trees are surveyed in [30].

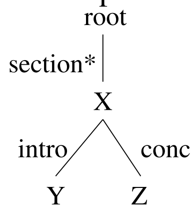
There is a *prima facie* mismatch between DTDs and tree automata: DTDs describe unranked trees, whereas classical automata describe binary trees. There are two ways around this. First, unranked trees can be encoded in a standard way as binary trees. Alternatively, the machinery and results developed for regular tree languages can be extended to the unranked case, as described in [15] (an extension for unranked infinite trees is described in [2]). Either way, one can prove a surprising and satisfying connection between specialized DTDs and tree automata: *they are precisely equivalent* [15, 56].

The equivalence of specialized DTDs and tree automata is a powerful tool for understanding XML schema languages. Properties of regular tree languages transfer to specialized DTDs, including closure under union, difference, complement, decidability of emptiness (in PTIME) and inclusion (in EXPTIME), etc. Moreover, automata techniques can yield algorithmic insight into processing DTDs. For example, the naive algorithm for checking validity of an XML document with respect to a specialized DTD is exponential in the size of the document (due to guessing specializations for labels). However, the existence of a bottom-up deterministic automaton equivalent to the specialized DTD shows that validity can be checked in linear time by a single bottom-up pass on the document. Efficient incremental validation of XML documents also makes use of tree automata [11]. Finally, as we will see shortly, tree automata are crucial tools in the static analysis of XML queries.

3. XML Queries

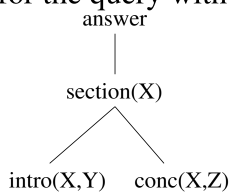
There has been a flurry of proposals for XML query languages, culminating with XQuery, adopted as a W3C standard. Earlier popular proposals include XML-QL [22], XSLT (W3C Web site), XMAS [12], XQL [57], XDuce [36, 37], and Quilt [19]. While a survey of the query languages for XML is beyond the scope of this paper, we discuss two main paradigms that are present in most proposals, including XQuery. In the first approach, a query consists of two parts: (i) a *pattern* used to extract bindings for a set of variables, and (ii) a *construct* clause indicating how to build the answer from the set of bindings found in (i). The pattern in (i) is in the spirit of conjunctive queries, except more navigational in flavor and extended with limited recursion. The pattern can be viewed as a "map" indicating how to reach the variables from the root or from each other by regular path expressions. Thus, the pattern is a tree of variables strongly connected to the root and labeled by regular path expressions. The data values associated to nodes can be explicitly compared, allowing to perform data joins. Variables in the pattern bind to nodes in the input that can reach each other by paths matching the regular expressions in the pattern. For example, consider again XML documents described by the DTD $\begin{array}{l} root : section; \\ section \to intro, section^*, conc \end{array}$

A query against such documents might use the pattern



The variable X binds to sections, Y binds to X's introduction, and Z to X's conclusion.

The *construct* clause specifies how to build the answer (a labeled tree) from the set of bindings. For example, a construct clause for the query with the above pattern might be:



This specifies an answer as follows. One node $section(X)$ is created for each binding of X. For each binding of X, Y, a node labeled $intro(X, Y)$ is created, and for each binding of X, Z a node $conc(X, Z)$ is created. (The expressions *section*(*X*), *intro*(*X,Y*), *conc*(*X,Z*) are referred to as *Skolem functions*, see [1].) For each binding of X, Y, Z, the nodes are linked as specified. So the answer consists of the unnested sequence of sections, and for each section its introduction and conclusion. Most languages allow nesting of queries within the *construct* clause.

Several subtleties distinguish XML from semi-structured data. First, it must be ensured that the output is a tree, so care must be taken in how links are specified. Second, the output must be an ordered tree, so mechanisms are needed for specifying the desired order. The order induced on the bindings by the input tree is usually the default. Also, some query languages (e.g. XMAS, YATL) allow querying the order of the input tree, by placing ordering conditions on variables in the pattern bound to sibling nodes, and even using horizontal path expressions among them.

In the approach described above, the depth of nesting of the *construct* clause is statically determined by the syntax of the query. This is a serious limitation. For example, one cannot make a simple change to an XML document, such as replacing all *name* labels occurring under *person* by *pname*, without knowing its structure. Intuitively, such a query has the flavor of a tree *transformation*, where the output is obtained by modifying recursively the input tree. The language XSLT allows defining such transformations (and much more complex ones!) using structural recursion on trees. For example, (using the notation $a(t_1, t_2)$ to denote a tree with root a and subtrees t_1, t_2), the above transformation can be expressed on binary trees as the function f defined by structural recursion as follows:

$$\begin{aligned} f(x(t_1,t_2)) &= \text{if } x \neq \textit{person} \text{ then } x(f(t_1), f(t_2)) \\ & \text{ else } x(g(t_1), g(t_2)) \\ g(x(t_1,t_2)) &= \text{if } x \neq \textit{name} \text{ then } x(g(t_1), g(t_2)) \\ & \text{ else } \textit{pname}(g(t_1), g(t_2)) \end{aligned}$$

The expressiveness of the query languages for XML is not easy to characterize. They appear to be a mix of useful but rather ad-hoc constructs, with both declarative and navigational features. They have limited recursion, in the style of Datalog chain queries. The common core is monotonic, but monotonicity is lost under minor variations in the use of regular path expressions. Some variants are not even closed under composition. Thus, the classes of queries expressed by the languages for XML appear to be rather idiosyncratic and to lack robustness. Nonetheless, we show next that there is a formal framework that subsumes the tree manipulation core of all XML languages: tree transducers.

XML queries and pebble tree transducers XML query languages take trees as input and produce trees as output. Despite their diversity, it turns out that their tree manipulation capabilities are subsumed by a single model of tree transducer, called *k-pebble transducer* [45]. This provides a uniform framework for measuring the expressiveness of XML languages, and it is instrumental in developing static analysis techniques. In Section 4 we will see how the transducers can be used for typechecking XML queries.

The k-pebble transducer uses up to k pebbles to mark certain nodes in the tree. Transitions are determined by the current node symbol, the current state, and by the existence/absence of the various pebbles on the node. The pebbles are ordered and numbered $1, 2, \ldots, k$. The machine can place pebbles on the root, move them around, and remove them. In order to limit the power of the transducer the use of pebbles is restricted by a stack discipline: pebbles are placed on the tree in order and removed in reverse order, and only the highest-numbered pebble present on the tree can be moved.

The transducer works as follows. The computation starts by placing pebble 1 on the root. At each point, pebbles $1, 2, \ldots, i$ are on the tree, for some $i \in \{1, \ldots, k\}$; pebble i is called the *current pebble*, and the node on which it sits is the *current node*. The current pebble serves as the head of the machine. The machine decides which *transition* to make, based on the following information: the current state, the symbol under the current pebble, and the presence/absence of the other $i-1$ pebbles on the current node. There are two kinds of transitions: *move* and *output* transitions. *Move* transitions are of four kinds: they can place a new pebble, pick the current pebble, or move the current pebble in one of the four directions *down-left, down-right, up-left, up-right* (one edge only). If a move in the specified direction is not possible, the transition does not apply. After each move transition the machine enters a new state, as specified by the transition.

An *output* transition emits some labeled node and does not move the input head. There are two kinds of output transitions. In a *binary* output the machine spawns two computation branches computing the left and right child respectively. Both branches inherit the positions of all pebbles on the input, and do not communicate; each moves the k pebbles independently of the other. In a *nullary* output the node being output is a leaf and that branch of computation halts.

Looking at the global picture, the machine starts with a single computation branch and no output nodes. After a while it has constructed some top fragment of the output tree, and several computation branches continue to compute the remaining output subtrees. The entire computation terminates when all computation branches terminate.

It turns out that all transformations over unranked trees over a given finite alphabet expressed in existing XML query languages (XQuery, XML-QL, Lorel, StruQL, UnQL, and a fragment of XSLT) can be expressed as k-pebble transducers. This does *not* extend to queries with joins on data values, since these require an infinite alphabet. However, k-pebble transducers can be easily extended to handle data values. Details, as well as examples, can be found in [45].

The k-pebble transducers generalize several known formalisms. Aho and Ullman [4] introduce *tree-walking* automata. These devices have a single head which can move up and down the tree, starting from the root. The set of tree languages accepted by a tree-walking automata is included in the set of regular tree languages, but it is a long-standing open problem whether the inclusion is strict [24]. The question whether k-pebble transducers can simulate all bottom-up transducers can be reduced to this open problem (in fact the two problems become equivalent, when $k = 1$). For the case of strings, the analog of tree-walking automata are precisely the two-way automata, which are known to express all regular languages.

String automata with a rather restricted form of k-pebbles are considered by Goberman and Harel [31]. They prove certain lower bounds in the gap of succinctness of the expressibility of such automata. Similarly, it turns out that the emptiness problem for k-pebble automata has a non-elementary lower bound.

Other models. Another transducer model for XML queries, called *query automaton*, is described in [54]. This work was the first to use MSO to study query languages for XML. Query automata, however, differ significantly from k-pebble transducers: they take an XML input tree and return a set of nodes in the tree. By contrast a k-pebble transducer returns a new output tree. Several abstractions of XML languages are studied in [42], and connections to extended tree-walking transducers with look-ahead are established. Various static analysis problems are considered, such as termination, emptiness, and usefulness of rules. It is also shown that ranges of the transducers are closed under intersection with *generalized DTDs* (defined by tree regular grammars). Tree-walking automata and their relationship to logic and regular tree languages are further studied in [53].

Another computation model for trees, based on *attribute grammars*, is considered in [51]. These capture queries that return sets or tuples of nodes from the input trees. Two main variants are considered. The first expresses all unary queries definable by MSO formulas. The second captures precisely the queries definable by first-order inductions of linear depth. Equivalently, these are the queries computable on a parallel random access machine with polynomially many processors. These precise characterizations in terms of logic and complexity suggest that attribute grammars provide a natural and robust querying mechanism for labeled trees.

To remedy the low expressiveness of pattern languages based on regular path expressions, a guarded fragment of MSO that is equivalent to MSO but that can be evaluated much more efficiently is studied in [52, 58]. For example, it is shown that this fragment of MSO can express FO extended with regular path expressions. In [14] a formal model for XSLT is defined incorporating features like modes, variables, and parameter passing. Although this model is not computational complete, it can simulate k-pebble transducers, even extended with equality tests on data values. Consequently, and contrary to conventional wisdom, XSLT can simulate all of XML-QL!

Feedback into automata theory. The match between XML and automata theory is very promising, but is not without its problems. The classical formalism sometimes needs to be adapted or extended to fit the needs of XML. For example, tree automata are defined for ranked trees, but XML documents are unranked trees. This required extending the theory of regular tree languages to unranked trees [15], and has given rise to a fertile line of research into formalisms for unranked trees. This includes extensions of tree transducers [42], push-down tree automata [46], attribute grammars [47], and caterpillar expressions [16]. Another mismatch arises from the fact that XML documents have data values, corresponding to trees over *infinite* alphabets. Regular tree languages over infinite alphabets have not been studied, although some investigations consider the string case [39, 55]. Tree-walking transducers accessing data values of XML documents are considered in [50].

XML schema languages contain new constructs allowing to specify flexible order constraints, and in particular to mix ordered and unordered data. XML query languages in turn provide constructs to specify the ordering of nodes in the answer. Neither aspect is captured by traditional tree automata and transducer models.

XML is already stimulating new research directions in language theory, and this trend is likely to amplify. A successful relationship will be a symbiotic one, in the mold of relational database theory and finite-model theory. Informative surveys on logic and automata-theoretic approaches to XML are provided in [49, 48].

4. Typechecking XML Queries

In relational databases, typechecking is a non-issue: in the standard relational query languages, the schema of the result is apparent from the syntax of the query. The situation is very different for XML. Whether the result of an XML query (or transformation) always satisfies a target XML Schema is far from obvious. Moreover, this is an important question in many scenarios. A typical one is data integration, where a user community would agree on a common schema and on producing only XML documents that are valid with respect to the specified schema.

The *(static) typechecking* problem is the following: given an input XML schema d (abstracted as a specialized DTD) a query q, and an output schema d', is it the case that
$q(sat(d)) \subseteq sat(d')$?

Related to the typechecking problem is the *type inference* problem[1]: given an input schema d and a query q, compute an output schema $\bar{q}(d)$ for $q(sat(d))$. This can mean several things. If $q(sat(d)) \subseteq sat(\bar{q}(d))$ then the inference algorithm computing $\bar{q}(d)$ is *sound*, and this is clearly a minimum requirement. Ideally, it would also be the case that $q(sat(d)) = sat(\bar{q}(d))$; then the inference algorithm is said to be sound and complete. Note that, in particular, a sound and complete inference algorithm would also solve the typechecking problem. Indeed, to verify

[1]The variant we state differs from that used in programming languages by assuming the input type is given.

that $q(sat(d)) \subseteq sat(d')$ it would be sufficient to check that $sat(\bar{q}(d)) \subseteq sat(d')$, which is decidable.

Unfortunately, sound and complete type inference is not possible for standard XML queries. An approach to incomplete type inference is taken by XDuce [36, 37]. In XDuce, types are essentially specialized DTDs. Recursive functions can be defined over XML data by pattern matching against regular expressions. XDuce performs static typechecking for these functions, verifying that the output of a function will always be of the claimed output type. However, the typechecking algorithm is only sound, not complete: one can write in XDuce a function that always returns results of the required output type, but that the typechecker rejects. This is expected in a general-purpose language that can express non-terminating functions. XDuce focuses on making the typechecker practical, both for the application writer and for the language implementer. A similar approach is taken by YATL [21, 20]. This language for semistructured data has an original type system, based on unordered types. Like XDuce, YATL admits incomplete type inference.

It turns out that sound and complete typechecking can be performed for a wide variety of XML languages so long as they query the tree structure of the input but not its data values. This is explored in [45] using the k-pebble transducer. As discussed earlier, this subsumes the tree manipulation core of most XML languages. Typechekcing can be done by means of *inverse type inference*. Suppose d is an input specialized DTD (or, equivalently, a tree automaton), and d' an output specialized DTD. Consider a k-pebble transducer T. It can be shown that $T^{-1}(sat(d'))$ is always a regular tree language, for which a tree automaton can be effectively constructed from T and d'. Then typechecking amounts to checking that $sat(d) \subseteq T^{-1}(sat(d'))$, which is decidable.

There are several limitations to the above approach. First, the complexity of typechecking in its full generality is very high: a tower of exponentials of height equal to the number of pebbles, so non-elementary. Thus, general typechecking appears to be prohibitively expensive. However, the approach can be used in restricted cases of practical interest for which typechecking can be reduced to emptiness of automata with *very few pebbles*. Even one or two pebbles can be quite powerful. For example, typechecking selection XML-QL queries without joins (i.e., queries that extract the list of bindings of a variable occurring in a tree pattern) can be reduced to emptiness of a 1-pebble automaton with exponentially many states. Another limitation has to do with data values. In general, the presence of data values leads to undecidability of typechecking. For example, if k-pebble transducers are extended with equality tests on the data values sitting under the pebbles, even emptiness is undecidable. However, the approach can be extended to restricted classes of queries with data value joins [5]. An overview of typechecking for XML is provided in [61].

Another twist in the typechekcing problem arises in the increasingly common scenario of relational databases exporting XML views of the data. Queries are then mappings from relations to trees. For example, SilkRoute is a research prototype enabling the definition of XML views from a relational database [27]. The typechecking problem now asks whether all views generated from the database satisfy a target DTD, possibly specialized. The database itself may satisfy given integrity constraints. This problem is investigated in [6], using an abstraction of the query language of SilkRoute. Once again, the general problem is undecidable, and the limits of decidability are established.

5. Other Research on XML

In this section we briefly mention some other recent theoretical research on XML, including constraints for XML, the complexity of XPath query evaluation and containment testing, validation of streaming XML, incremental XML validation, and XML with incomplete information.

Constraints in XML Constraints are essential ingredients to classical databases. While their primary role is as a filter of invalid data, they are also useful in query optimization, schema design, and choice of efficient storage and access methods. The most common database constraints are *functional dependencies* (fds) and *inclusion dependencies* (incds). Not surprisingly, these continue to be important in XML. Indeed, there is a natural need to express inclusion dependencies in XML documents. In addition, key constraints are part of various schema proposals, such as XML Schema. Both types of constraints also arise in XML documents that are generated from databases.

In XML, both key constraints and inclusion dependencies involve the *data values* associated to the leaves of XML documents (or to values of attributes viewed as leaf elements). Inclusion dependencies in XML can be expressed using path expressions, with extensions for the non-unary case [26]. Key constraints can be formalized as a pair $(q, \{p_1, \ldots, p_n\})$ where q and the p_i's are path expressions. Intuitively, q identifies the elements e to which the key constraint applies and $p_1, \ldots p_n$ the nodes whose data values collectively identify each element e. More precisely, if e, f are nodes reachable from the root by paths in q, the node e_i is reachable from e by a path in p_i, the node f_i is reachable from f by a path in p_i, and the values of e_i and f_i are equal, $1 \leq i \leq n$, then e and f are the same node. Note that this definition uses separate notions of value equality and node equality.

The implication problem for key constraints as above is harder than in the relational case, because it involves reasoning about regular path expressions (and recall that equivalence of regular expressions in isolation is already PSPACE-hard [29]). Restrictions on the path expressions leading to an $O(n^2)$ algorithm for testing implication are shown in [17].

There is an intricate interaction between XML constraints and DTDs. As shown in [25], the satisfiability problem for key and foreign key constraints becomes undecidable in the presence of DTDs (and is NP-complete in the unary case), whereas it is trivial in classical databases. Checking consistency of DTDs together with constraints is further studied in [8], and normal forms for XML documents are discussed in [9, 10]. The impact of DTDs and other schema formalisms on constraints is interesting both theoretically and practically, and remains largely unexplored. A survey of constraints in semi-structured data and XML is presented in [18]. Constraints in semi-structured data are also discussed in [1].

The complexity of query evaluation and containment The complexity of XML query evaluation, especially the heavily used XPath fragment, is of significant practical and theoretical interest. The first polynomial-time algorithm for XPath processing (with respect to combined complexity) was proposed only recently [32], using a dynamic programming technique. Even to this day all major XPath engines take time exponential in the size of the input queries (see [33]). From a theoretical standpoint, the precise complexity of XPath query evaluation is open, and it is thus unknown whether the query evaluation problem can be parallelized. In [33] it is shown that both the data complexity and the query complexity of XPath 1.0 fall into low (highly parallelizable) complexity classes, but that the combined complexity is PTIME-hard. A large and practically important fragment of XPath 1.0 is identified, for which the combined complexity is LOGCFL-complete and, therefore, in the highly parallelizable complexity class NC^2. The complexity of XPath query evaluation is also studied in [59].

The containment and equivalence of XML queries are of paramount importance to query optimization. The containment problem for conjunctive queries with regular expressions is investigated in [28]. The complexity of the containment and equivalence problem for various fragments of the XPath queries is investigated in [23, 62, 43]. In particular, it is shown in [43] that containment of XPath queries with branching, label wildcards, and descendant relationships between nodes is CO-NP complete, although it is in PTIME for any combination of two of these features [7, 44]. A sound containment algorithm that is efficient in all cases but may return false negatives is also provided in [43]. The minimization of tree patterns is examined in [7] and minimization of XPath expressions is studied in [63].

Validating streaming XML A line of research related to typechecking is the on-line validation of streaming XML documents with respect to a DTD, under memory constraints (e.g., see [38, 41, 60]). Validation using constant memory, formalized by a finite-state automaton (FSA), is considered in [60]. Two flavors of the problem are examined, depending on whether or not the XML document is assumed to be well-formed. The main results provide conditions on the DTDs under which validation of either flavor can be done using an FSA. For DTDs that cannot be validated by an FSA, two alternatives are investigated. The first relaxes the constant memory requirement by allowing a stack bounded in the depth of the XML document, while maintaining the deterministic, one-pass requirement. The second approach consists in refining the DTD to provide additional information that allows validation by an FSA. Formalizing this requires automata and transducer models that perform a single traversal of the input tree in depth-first, left-to-right order.

A problem related to the above is the processing of XPath filters applied to streaming XML documents. Satisfaction of a filter is a Boolean query that, theoretically, can be viewed as a special case of checking validity with respect to a specialized DTD. However, the simultaneous monitoring of large numbers of XPath filters raises specific optimization issues. The efficient processing of such filters using automata techniques is investigated in [34, 35].

Incremental XML validation When an XML document is updated, it has to be verified that the new document still satisfies its DTD or XML Schema. Doing this efficiently is a challenging problem that is central to many applications. Brute-force validation from scratch is often not practical, because it requires reading and validating the entire database following each update. Instead, it is desirable to develop algorithms for incremental validation. The efficient incremental validation of XML documents with respect to DTDs and XML Schemas, under updates consisting of element tag renamings, insertions and deletions, is studied in [11]. For DTDs, an $O(m \log n)$ incremental validation algorithm is shown that uses an auxiliary structure of size $O(n)$, where n is the size of the document and m the number of updates. A restricted class of DTDs called "local" is also considered. Intuitively, these are DTDs using regular expressions for which validation after an update can be done locally, by examining just substrings at bounded distance from the position of the update. As shown in [11], such DTDs arise very frequently and can be incrementally validated in practice in constant-time data complexity with $O(\log n)$ space overhead consisting of 4-byte integer counters. For specialized DTDs, an $O(m \log^2 n)$ incremental algorithm is provided, using an auxiliary structure of size $O(n)$. This is a significant improvement over brute-force re-validation from scratch.

XML with incomplete information The representation and querying of XML documents with incomplete information is considered in [3]. They introduce a simple model for XML data and their DTDs, a very simple query language, and a representation system for incomplete information in the spirit of the representations systems developed by Imielinski and Lipski for relational databases. In the scenario considered, the incomplete information about an XML document is continuously enriched by successive queries to the document. It is shown that the representation system can capture the partial information about the source document acquired by successive queries, and that it can be used to intelligently answer new queries. The impact on complexity of enriching the representation system or query language with additional features is also considered. The results suggest that the proposed approach achieves a practically appealing balance between expressiveness and tractability. The immediate motivation for this research was the Xyleme project at INRIA, whose objective it to develop a data warehouse for Web XML documents.

6. Conclusion

We presented some of the recent developments in the rich theory emerging around XML and its schema and query languages. We focused mainly on powerful and elegant connections between XML schema and query languages and tree automata theory, and their applications to typechecking. A few other areas of research were only briefly described. They include XML constraints and normal forms, XPath query evaluation, query containment and minimization, processing streaming XML, incremental validation of XML documents, and the representation and querying of XML documents with incomplete information. Due to space limitations, other important research directions were left out of this incomplete account.

It is apparent that, with some delay, the theory of XML is catching up with the practice. It is fast emerging as a vigorous area of research, with significant intellectual content and potential impact on applications. With some effort and luck, the relationship between the theory and practice of XML will become more symbiotic than in XML's early days.

References

[1] S. Abiteboul, P. Buneman, and D. Suciu. *Data on the Web.* Morgan Kauffman, 1999.

[2] S. Abiteboul and P. C. Kanellakis. Object identity as a query language primitive. *JACM*, 45(5):798–842, 1998. Extended abstract in SIGMOD'89.

[3] S. Abiteboul, L. Segoufin, and V. Vianu. Representing and querying XML with incomplete information. In *Proc. ACM PODS*, 2001.

[4] A. Aho and J. Ullman. Translations on a context free grammar. *Information and Control*, 19(19):439–475, 1971.

[5] N. Alon, T. Milo, F. Neven, D. Suciu, and V. Vianu. XML with data values: typechecking revisited. In *Proc. ACM PODS*, 2001.

[6] N. Alon, T. Milo, F. Neven, D. Suciu, and V. Vianu. Typechecking XML views of relational databases. *ACM Transactions on Computational Logic*, 4(3):315–354, 2003. Extended abstract in LICS 2001.

[7] S. Amer-Yahia, S. Cho, V. S. Lakshmanan, and D. Srivastava. Minimization of tree pattern queries. In *Proc. SIGMOD Conf.*, 2001.
[8] M. Arenas, W. Fan, and L. Libkin. On verifying consistency of XML specifications. In *Proc. ACM PODS*, pages 259–270, 2002.
[9] M. Arenas and L. Libkin. A normal form for XML documents. In *Proc. ACM PODS*, pages 85–96, 2002.
[10] M. Arenas and L. Libkin. An information-theoretic approach to normal forms for relational and XML data. In *Proc. ACM PODS*, pages 15–26, 2003.
[11] A. Balmin, Y. Papakonstantinou, and V. Vianu. Incremental validation of XML documents. *ACM TODS*. To appear. Extended abstract in ICDT 2003.
[12] C. Baru et al. XML-based information mediation with MIX. In *ACM SIGMOD Conf. Demo.*, pages 597–599, 1999.
[13] C. Beeri and T. Milo. Schemas for integration and translation of structured and semi-structured data. In *Int'l. Conf. on Database Theory*, pages 296–313, 1999.
[14] G. Bex, S. Maneth, and F. Neven. A formal model for an expressive fragment of XSLT. In *Proc. DOOD*, pages 1137–1151, 2000.
[15] A. Bruggemann-Klein, M. Murata, and D. Wood. Regular tree and regular hedge languages over unranked alphabets, 2001. Technical Report HKUST-TCSC-2001-0, Hong-Kong University of Science and Technology.
[16] A. Brüggemann-Klein and D. Wood. Caterpillars: a context specification technique. *Markup Languages*, 2(1):81–106, 2000.
[17] P. Buneman, S. Davidson, W. Fan, C. Hara, and W. Tan. Keys for XML. In *Proc. WWW-10*, 2001.
[18] P. Buneman, W. Fan, J. Simeon, and S. Weinstein. Constraints for semi-structured data and XML. *SIGMOD Record*, 30(1), 2001.
[19] D. Chamberlin, J. Robie, and D. Florescu. Quilt: An XML query language for heterogeneous data sources. In *WebDB (Informal Proceedings)*, pages 53–62, 2000.
[20] V. Christophides, S. Cluet, and J. Simeon. On wrapping query languages and efficient XML integration. In *Proc. ACM SIGMOD*, pages 141–152, 2000.
[21] S. Cluet, C. Delobel, J. Simeon, and K. Smaga. Your mediators need data conversion! In *Proc. ACM SIGMOD Conf.*, pages 177–188, 1998.
[22] A. Deutsch, M. Fernandez, D.Florescu, A. Levy, and D. Suciu. A query language for XML. In *WWW8*, pages 11–16, 1999.
[23] A. Deutsch and V. Tannen. Containment and integrity constraints for XPath fragments. In *Proc. KRDB*, 2001.
[24] J. Engelfriet, H. Hoogenboom, and J. Best. Trips on trees. *Acta Cybernetica*, 14:51–64, 1999.
[25] W. Fan and L. Libkin. On XML integrity constraints in the presence of DTDs. *JACM*, 49(3):368–406, 2002.
[26] W. Fan and J. Siméon. Integrity constraints for XML. In *Proc. ACM PODS*, pages 23–34, 2000.
[27] M. Fernandez, W. Tan, and D. Suciu. Silkroute: trading between relations and XML. *Computer Networks*, (33):723–745, 2000.
[28] D. Florescu, A. Levy, and D. Suciu. Query containment for conjunctive queries with regular expressions. In *Proc. ACM PODS*, pages 139–148, 1998.
[29] M. R. Garey and D. S. Johnson. *Computers and Intractability*. Freeman, 1979.

[30] F. Gécseg and M. Steinby. Tree languages. In G. Rozenberg and A. Salomaa, editors, *Handbook of Formal Languages*, volume 3, chapter 1, pages 1–68. Springer, 1997.

[31] N. Globerman and D. Harel. Complexity results for two-way and multi-pebble automata and their logics. *TCS*, 169(2):161–184, 1996.

[32] G. Gottlob, C. Koch, and R. Pichler. Efficient algorithms for processing XPath queries. In *Proc. VLDB*, pages 95–106, 2002.

[33] G. Gottlob, C. Koch, and R. Pichler. The complexity of XPath query evaluation. In *Proc. ACM PODS*, pages 179–190, 2003.

[34] T. Green, G. Miklau, M. Onizuka, and D. Suciu. Processing XML streams with deterministic automata. In *Proc. ICDT*, pages 173–189, 2003.

[35] A. Gupta and D. Suciu. Stream processing of XPath queries with predicates. In *Proc. ACM SIGMOD*, pages 419–430, 2003.

[36] H. Hosoya and B. Pierce. XDuce: A typed XML processing language (Preliminary Report). In *WedDB (Informal Proceedings)*, pages 111–116, 2000.

[37] H. Hosoya, J. Vouillon, and B. Pierce. Regular expression types for XML. In *Int. Conf. on Functional Programming*, pages 11–22, 2000.

[38] Z. Ives, A. Levy, and D. Weld. Efficient evaluation of regular path expressions on streaming XML data. Univ. of Washington Tech. Rep. CSE000502.

[39] M. Kaminski and N. Francez. Finite-memory automata. *Theoretical Computer Science*, 134(2):329–363, 1994.

[40] D. Lee and W. Chu. Comparative analysis of six XML schema languages. *SIGMOD Record*, 29(3):76–87, 2000.

[41] B. Ludaescher, P. Mukhopadhyay, and Y. Papakonstantinou. A transducer-based XML query processor. In *Proc. VLDB Conf.*, 2002.

[42] S. Maneth and F. Neven. Structured document transformations based on XSL. In *Proc. DBPL*, pages 79–96. LNCS, Springer, 1999.

[43] G. Miklau and D. Suciu. Containment and equivalence for an XPath fragment. In *Proc. ACM PODS*, pages 65–76, 2002.

[44] T. Milo and D. Suciu. Index structures for path expressions. In *Proc. ICDT Conf.*, 1999.

[45] T. Milo, D. Suciu, and V. Vianu. Typechecking for XML transformers. *J. of Computer and System Sciences*, 66(1):66–97, 2003. Extended abstract in PODS 2000.

[46] A. Neumann and H. Seidl. Locating matches of tree patterns in forests. In *Proc. Foundations of Software Technology and Theoretical Computer Science*, pages 134–145. LNCS, Springer, 1998.

[47] F. Neven. Extensions of attribute grammars for structured document queries. In *Proc. DBPL*, pages 97–114. LNCS, Springer, 2000.

[48] F. Neven. Automata, logic, and XML. In *Proc. Computer Science Logic*, pages 2–26. Springer LNCS, 2002.

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

[50] F. Neven. On the power of walking for querying tree-structured data. In *Proc. ACM PODS*, pages 77–84, 2002.

[51] F. Neven and J. V. den Bussche. Expressiveness of structured document query languages based on attribute grammars. *JACM*, 49(1), 2002. Extended abstract in PODS 1998.

[52] F. Neven and T. Schwentick. Expressive and efficient pattern languages for tree-structured data. In *Proc. ACM PODS*, pages 145–156, 2000.

[53] F. Neven and T. Schwentick. On the power of tree-walking automata. In *Proc. ICALP*, pages 547–560, 2000.

[54] F. Neven and T. Schwentick. Query automata on finite trees. *Theoretical Computer Science*, 275(1-2):633–674, 2002.
[55] F. Neven, T. Schwentick, and V. Vianu. Towards regular languages over infinite alphabets. In *Proc. MFCS*, pages 560–572, 2001.
[56] Y. Papakonstantinou and V. Vianu. DTD inference for views of XML data. In *Proc. ACM PODS*, pages 35–46, 2000.
[57] J. Robbie, J. Lapp, and D. Schach. XML query language (XQL). In *The Query Languages Workshop (QL'98)*, 1998.
[58] T. Schwentick. On diving in trees. In *Proc. MFCS*, pages 660–669, 2000.
[59] L. Segoufin. Typing and querying xml documents: some complexity bounds. In *Proc. ACM PODS*, pages 167–178, 2003.
[60] L. Segoufin and V. Vianu. Validating streaming XML documents. In *Proc. ACM PODS*, pages 53–64, 2002.
[61] D. Suciu. The XML typechecking problem. *SIGMOD Record*, 31(1):89–96, 2002.
[62] P. Wood. On the equivalence of XML patterns. In *Proc. DOOD*, pages 1152–1166, 2000.
[63] P. Wood. Minimizing simple XPath expressions. In *Proc. WebDB*, 2001.