XML: Model, Schemas, Types, Logics and Queries

Nils Klarlund[1], Thomas Schwentick[2], and Dan Suciu[3]

[1] AT&T Labs – Research,USA
email: klarlund@research.att.com
[2] University of Marburg, Germany
email: tick@Mathematik.Uni-Marburg.de
[3] University of Washington, USA
email: suicu@cs.washington.edu

1 Introduction

It is intriguing that something as bland as a syntax for trees has become one of the leading buzzwords of the Internet era. *XML* (eXtended Markup Language) is the name of this notation, which evolved from document processing needs. This mystery is deepened by the apparent antiquity of the central idea – that linear syntax can represent trees. *S-expressions*, invented forty years ago, are at least as general, but never became the universally accepted foundation for data interchange and programming, despite their enormous influence in programming language theory and artificial intelligence research. Additionally, trees have been studied intensively in computer science ever since, so we might suspect that the reason for the excitement is simply that practitioners are catching up with methods of abstraction and representation via trees that are well-known in academia.

In this chapter, we shall see that the suspicion is easy dispelled. We look at techniques now used in practice for dealing with XML trees, and we note how they depart from old-fashioned uses. Since trees are objects that are very complicated to manipulate directly through pointer updates, declarative techniques are becoming increasingly important, especially when it comes to exploring, mining, and constructing tree-shaped data. In particular, we will contrast conventional concepts of database theory such as relational calculus with that of more procedural notations for trees. We explore why the essential problem of locating data in trees is intimately linked with tree automata and decidable logics, somewhat in parallel to the link between query algebras and first-order logic in relational database theory. So, we shall see why logic and automata create interesting new research opportunities.

1.1 XML: An Important Practical Paradigm

XML [28] is a convention for representing *labeled* or *tagged* trees as a text, that is, as a sequence of characters. In essence, the representation is a Dyck language, where the text corresponding to a node is delimited by beginning and ending parentheses that each mentions the label. The text between these parentheses describes the children of the node.

XML is in essence a simplified form of *SGML* (Standard Generalized Markup Language) [44], a notation that has been in existence for number of years in the specialized area of professional document preparation. The W3C (World Wide Web Consortium, `www.w3.org`) launched HTML, a specific SGML markup vocabulary, as the document standard that soon would carry most information on the Internet. Later, the W3C saw the potential of a data and content distribution format that would be more general, and it oversaw the pruning of SGML into XML. Most XML-related core standards are a result of work done under the auspices of the W3C, although we shall also discuss alternative technology that has emerged as a result of the perceived unnecessary complexity of essential XML specifications.

The labeling of data that is inherent to XML has led to the frequent characterization of XML data as "self-describing." Usually, this is an exaggeration: the "description" is merely the labeling of the tree, something that by itself carries no meaning to neither humans nor machines. Nevertheless, the tagging of data is essential to a main application of XML: the representation of irregular data that do not easily conform to the tabular view of relational databases. The labeling enables irregularities to be amenable to automated processing. As an example, assume that information tagged `book` has a certain subtree structure, including descendants named `author` and `title`. Then such a subtree may be inserted flexibly in various structures that need to mention books. And, applications can easily identify the subtree structure of a book. In general, there is, of course, a system to the irregularities. The set of possible structures are defined in notations called *schema languages*. These notations define the types of nodes and their relations in terms of their labels.

In general, labeled graphs for representation of information are called *semi-structured data*. XML is superficially less general, since it emphasizes trees. But just as the relational data model can represent arbitrary graphs by using attribute values as identifiers, XML allows arbitrary cross edges, something anticipated at the most fundamental level.

1.2 Overview of this Chapter

We conclude this introduction with some motivating examples of XML applications as found in current and emerging business practice. Then, in Sect. 2 we provide a walk-through of the fundamentals of XML. We discuss the syntax and present a mathematical model that succinctly summarizes recent official W3C attempts at capturing the essence of XML. Next, in Sect. 3, we discuss already deployed schema notations such as *DTDs*, *RELAX NG*, and *XML Schema*, and we hint at some interesting theoretical issues. In Sect. 4, we introduce the recent program notations that are of key practical significance:

XPath for locating nodes in an XML document, *XSLT* for transforming documents, and *XQuery* for information extraction à la SQL. Following these practical aspects of XML, we turn to the theory of regular tree languages in Sect. 5 since regularity and its connection to tree automata and logic provide us with a set tools for understanding XML notations. In Sect. 6, we relate automata and logic to XML query languages with particular emphasis on the type checking problem. Finally, in Sect. 7, we mention some other promising research directions.

1.3 XML Applications

Our theoretical interest in XML is supported by several classes of applications which we briefly discuss below.

XML Documents The simplest and original application is to use XML for document markup. In these applications XML documents are relatively small, and their primary usage is to be rendered by a browser; alternatively, they may first be translated by an XSLT program into a format readable by a browser.

The tree paradigm in document practice is illustrated by the data-driven transformations of content that any modern browser implements. For example, with *CCS* (*Cascading Style Sheets* [12]), a programmer has a simple, declarative way of characterizing visual layout of text marked up in HTML or XML as a function of the location of nodes in the document tree. The more powerful transformation notation XSLT, based on a tree-walking paradigm, is a Turing-complete functional programming language that is also built into some modern browsers. Such notations are key instruments for achieving the abstractions that put a wedge between content and layout.

XML in Data Exchange One of the greatest promises of XML is to serve as the lingua franca in data exchange. Business data, scientific data, medical data, virtually any data that is manipulated today by some application can be translated to XML and exchanged directly between applications belonging to different businesses or organizations. While Internet applications require much more than the ability to exchange data, our focus is on description, retrieval, and transformation of the data as it is exchanged. The common thread is that different organizations would agree on a common schema for the XML being exchanged, then would design applications that generate XML data obeying the schema, and then consume the data described in the schema.

For example, a telecommunications company may offer a service allowing retail companies to set up their Web sites, including product catalogs. Such a Web site is described most abstractly through XML documents that summarize product descriptions through words, pictures, and links to related products – along with various information about the company. A description of the allowed and required data in a sufficiently simple formal notation becomes a tool for describing the service to retail companies and for writing contracts. Later, the same descriptions are used by programmers of the retail company to make sure the parametrizations they supply are correct. Although many other system description languages exists, XML is uniquely useful for such loosely structured and abstract data.

XML data is often generated from other data sources. For example the XML data might be created from a relational database. A more interesting piece of technology is a *query translator*. It may rewrite XQuery expressions into SQL expressions. Recent research has focused on the problem of optimizing the resulting SQL queries, which tend to be complex and unusual for traditional query optimizers. Since XML is almost always generated dynamically in this application, and needs to conform to a given schema, typechecking is expected to play a key role.

Native XML Data Pushing this idea even further, some argue to design new business applications using only XML data. Some companies are currently building native XML database systems, with an XQuery processor. Traditional query processing technologies apply here (query optimization, data statistics, etc), and need to be adapted and extended to the richer XML data model. A more conservative approach is to use a relational database system to store and process XML data. This requires technologies for storing XML data in relational databases, and for translating XML queries into SQL queries.

XML Messaging For financial service applications, large amounts of event-driven XML content is to be published and distributed in order to be received instantaneously by subscribers according to personalized criteria. Thus arises the idea of XML *content-based routing* [76], where such messages are filtered and forwarded in a network based on their content. Routing can be defined through XPath expressions, which check conditions on the XML packets. A router stores a large number of boolean XPath expressions that are evaluated on each incoming XML document. The document is then forwarded to other routers or to clients based on the result of the evaluation of those XPath expressions. Sometimes the XML messages are subjected to simple transformations, such as extracting or deleting certain fields, or, say, conversion from ASCII to Unicode. The key technology needed is the ability to process efficiently large numbers of XPath expressions on a stream of XML documents. More advanced technologies require management of large collections of XPath expressions: adding, deleting an XPath expression, testing two expressions for equality or for containment, etc.

2 XML Fundamentals

We introduce XML through examples. Then, we provide a formal definition of XML that forms the basis for the logical foundation of XML in this chapter.

2.1 XML Through Examples

We present the syntax for a case that is both database and document-oriented, a common situation for content on the Internet. Our example is a cooking recipe, which contains both text and structural information. A recipe in XML might look as in Fig. 1.

```
<recipe name="My first cake" estimated-time="1" unit="h">
  <ingredients>
    <ingredient quantity="200" unit="g" name="flour"/>
    <ingredient quantity="100" unit="g" name="sugar"/>
    <ingredient quantity="2" unit="dl" name="milk"/>
  </ingredients>
  <procedure>Throw all the ingredients in a bowl and stir
     <em>vigorously</em>.  Pour resulting batter into a pan and bake
     for thirty minutes.
  </procedure>
</recipe>
```

Fig. 1. An XML document: the recipe example.

Illustrating all the key concepts of XML syntax, we explain this snippet as follows. A *recipe* is characterized by the *attributes* name, estimated-time, and unit. Each attribute has a string value. The balanced, labeled parentheses <recipe>...</recipe> are called an *element*, which is the linear syntax for a node labeled *recipe*. The beginning parenthesis <recipe> is called a *start tag* and the end parenthesis </recipe> is called an *end tag*. The inside of <recipe>...</recipe> is called the *content* of the node; it is a linear notation for its *children*, which are ordered. In this case, there are two children, corresponding to the elements named ingredients and procedure. The attributes also correspond to named direct descendants, but they are not ordered and they are not usually regarded as children. The content of the procedure element represents three children: a *text node* whose value is "Throw␣all␣the␣ingredients␣in␣a␣bowl␣and␣stir\n␣␣␣␣␣", where ␣ denotes a space and \n denotes the line separator. The second node is the one named em; it contains the text that, presumably, is supposed to be emphasized. The third node is the remaining text, again including the white space characters. This kind of data interspersed in text and elements is called *mixed content*.

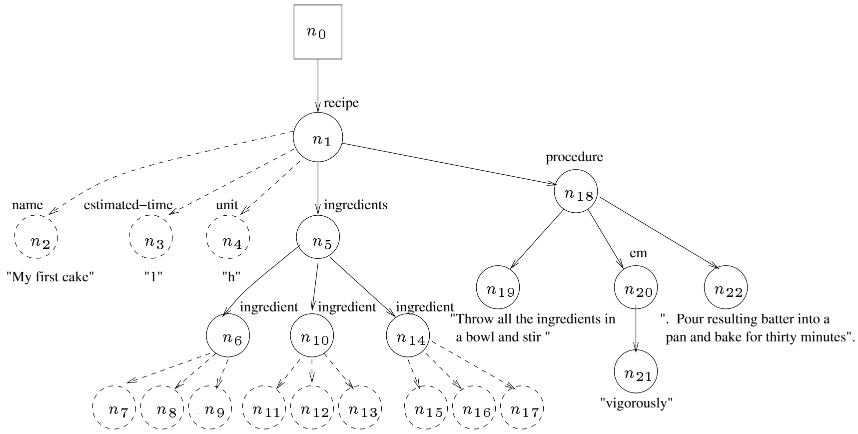

Fig. 2. The tree corresponding to the recipe document in Fig. 1. Some details are omitted. For example, the nodes n_7, n_8, and n_9 are the attribute nodes named `quantity`, `unit`, and `name` of the first `ingredient`.

The `ingredient` elements have no content, but only attributes; the "/" character after the name indicates that this tag is both a start and end tag at the same time.

Every XML document can be represented as a tree, with nodes corresponding to elements, attributes, and text values. Fig. 2 illustrates the tree for the XML document in Fig. 1 with some details omitted. The node n_1 corresponds to the element `<recipe> ...</recipe>`; node n_4 corresponds to the attribute `unit="h"`; and node n_{21} corresponds to the text `"vigorously"`. Note that there is a *root node* n_0, which has no corresponding element or attribute in the XML document and is called the *document node*.

The reader may be puzzled: apparently the characters between tags sometimes denote text nodes and sometimes not. For example, the text between `<ingredients>` and `<ingredient>` consists of a newline character and several spaces, but there is no such text node in the tree; on the other hand, the text between the `<procedure>` tag and the `<em>` tag is turned into a text node. The reason is that in the tree depiction we have pruned all text nodes that are not appropriate for the intended data model. So, the tree denoted by the document above is really more complex than we would like; there are spurious text nodes almost everywhere.

The problem is that the document does not provide any information about which white space is significant and which is not. It may seem like a trivial point, but in practice, even experienced programmers are struggling with white space issues. That "self-describing" XML cannot describe even its own whitespace is perhaps ironic. In practice, the problem is solved through schemas that describe the data or through explicit programming.

quantity	unit	name
200	g	flower
100	g	sugar
2	dl	milk

```
<row quantity="200" unit="g" name="flour"/>
<row quantity="100" unit="g" name="sugar"/>
<row quantity="2" unit="dl" name="milk"/>
```

Fig. 3. An example of a relational database instance and the corresponding representation in XML.

The Relational Model vs. XML In the dominant relational approach to databases, all information is represented in tables. Let us compare the two approaches through our example. There the `ingredients` information can obviously be brought in tabular form since each `ingredients` node has the same three unordered attributes. Thus, the table (left) and the XML document (right) in Fig. 3 are almost equivalent. The main difference is that the XML representation imposes an ordering of the rows. We also note that the XML model carries around labels, even where we may not want them under the relational view. As indicated, we make the arbitrary convention that `row` is the canonical label of a tuple.

Semistructured Data XML can, of course, represent more than relational data. The kind of data it can express is called *semistructured data*. To explain it, consider an alternative XML representation of the relation above:

```
<row>
    <quantity>200</quantity>
    <unit>g</unit>
    <name>flour</name>
</row>
<row>
    <quantity>100</quantity>
    <unit>g</unit>
    <name>sugar</name>
</row>
<row>
    <quantity>2</quantity>
    <unit>dl</unit>
    <name>milk</name>
</row>
```

We simply migrated the data values from attributes to subelements (this makes the analogy easier with some irregular data shown below). This data is regular, or *structured*: each `row` has exactly one `quantity`, one `unit` and one `name`. The corresponding tree representation is shown in Fig. 4. Let us contrast this with the following XML instance:

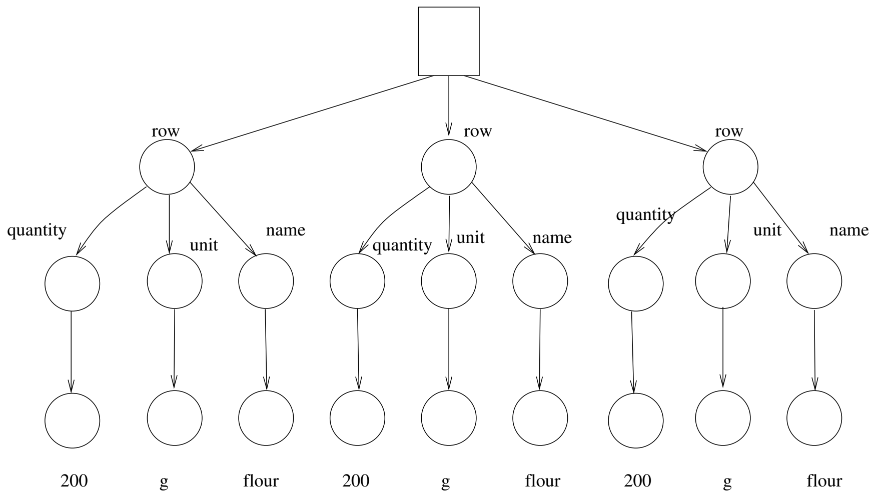

Fig. 4. The tree corresponding to the `ingredients` relation of the alternative XML represenation.

```
<row>
    <quantity>1</quantity>
    <name>apple</name>
</row>
<row>
    <quantity>1</quantity>
    <unit>count</unit>
    <name>apple</name>
    <name>banana</name>
</row>
<row>
    <quantity>200</quantity>
    <unit>g</unit>
    <name>
        <product>Corn Flakes</product>
        <manufacturer>CornINC</manufacturer>
    </name>
</row>
```

The first `row` has a missing `unit`, while the second `row` has two `names`. Finally, the third `row` has a `name` that is a nested structure, rather than being an atomic value. This example illustrates the main points in semistructured data: data items may have missing attributes, multiple occurrences of the same attribute, or attributes may have different types in different items.

Self-describing Data Consider the relational data in Fig. 3: `quantity` and `unit` are *schema* components, while `200` and `flower` are *data values*. This distinction is crucial in a relational system, which stores schema components in the system's catalog, separately from the data values. In fact, the *data* is stored in binary format and cannot be interpreted without the information from the catalog. In contrast, XML data is *self-describing* since the schema components are interleaved with data values. Schema components now have to occur multiple times, like `quantity`, making the format more verbose, but at the same time more flexible.

2.2 XML Model

What are the semantics of XML? For this question of what XML really is, a perplexing number of models have been proposed. Departing somewhat from the usual very simplistic view taken by XML theoreticians, we base our terminology on the *XQuery 1.0 and XPath 2.0 Data Model* [39], which is currently under development. Our model emphasizes the graph-theoretic aspect of trees, a point of view important to program interfaces. In fact, the standard programming model of XML, called *DOM* (*Domain Object Model*), is a close realization of this model: it treats nodes as objects and edges as object references, that is as pointers.

We assume that we have at our disposition two sets: a finite alphabet of *labels*, Σ, and an infinite alphabet of *data values*, D. Labels model XML tags and attributes. The XML standard rigidly specifies allowed character encodings, including Unicode, and the ways labels and data are made from characters. Our restriction of Σ to a finite set is needed for the development of the logical formalisms in Sect. 6 and is consistent with practice, since tags and attributes are usually drawn from a given schema. By contrast the set of data values, D, is infinite and includes strings, integers, dates, etc.

We can now formally define an XML tree.

Definition 1. An *XML tree* $t = (N, E, <, \lambda, \nu)$ consists of a directed tree (N, E), with N the set of *nodes* and $E \subseteq N \times N$ the set of *edges*, a total order $<$ on N, a partial *labeling function* $\lambda : N \to \Sigma$, and a partial *value function* $\nu : N \to D$; moreover, the order must be *depth-first*: whenever x is an ancestor of y, it holds that $x < y$.

We denote an XML tree as $t = (N, E)$, whenever λ, ν, and $<$ are understood from the context, and we sometimes denote N as $dom(t)$ and call it the *domain* of t. We let $\mathcal{T}_{\Sigma,D}$ denote the set of XML trees with labels from Σ and data values from D, and $\mathcal{T}_{\Sigma}$ denote the set of XML trees with labels from Σ only (i.e., ν is totally undefined). Each tree $t \in \mathcal{T}_{\Sigma,D}$ can be viewed as a tree in $\mathcal{T}_{\Sigma}$ by simply "forgetting" the data values: in other words, there exists a canonical function $\mathcal{T}_{\Sigma,D} \to \mathcal{T}_{\Sigma}$.

We shall adopt this simplified definition of XML trees in our discussion of XML's logical foundations. The reader should be warned that this is only an approximation of the W3C standard. More precisely, each XML document according to the standard can be uniquely converted into an abstract XML tree $t = (N, E)$. Figures 2 and 4 illustrate two examples of such trees. But the converse is not true: there are abstract trees that do not correspond to legal XML documents: for example data values must only be attached to leaves. It is possible to list all conditions that would make every tree $t = (N, E)$ correspond to an XML document. We list here these conditions for completeness, but will not use them in the rest of this chapter. The conditions are:

- N is partitioned into four disjoint sets: $N = \{d\} \cup \mathit{Elmt} \cup \mathit{Attr} \cup \mathit{Txt}$, where
  - d is the root of t and is called the *document node*.
  - *Elmt* is called the set of *element nodes*.
  - *Attr* is the set of *attribute nodes*.
  - *Txt* is called the set of *text nodes*.
- $\mathit{Attr} \cup \mathit{Txt}$ consists of leaf nodes only: if $(n, n') \in E$, then $n \in \{d\} \cup \mathit{Elmt}$.
- Attribute children precede all other types of children: formally, if $(n, n_1) \in E, (n, n_2) \in E$, $n_1 \in \mathit{Attr}$, and $n_2 \in \mathit{Elmt} \cup \mathit{Txt}$, then $n_1 < n_2$.
- The *labeling function* $\lambda : N \to \Sigma$ is defined on $\mathit{Elmt} \cup \mathit{Attr}$ and is undefined on all other nodes.
- The *value function* $\nu : N \to D$ is defined on $\mathit{Attr} \cup \mathit{Txt}$ and is undefined on all other nodes.
- Children of the document node d must be element nodes: if $(d, n) \in E$, then $n \in \mathit{Elmt}$.

A more subtle issue in the standard W3C XML specification is that the attribute nodes that are siblings are unordered. That is, an element may have several attribute, element, and text children, and the order relation $<$ has to place all attribute nodes before the element and text nodes, but the attributes are not ordered among themselves. Thus, the order relation $<$ is in reality a partial relation. Definition 1 does not capture this state of affairs; instead it defines $<$ to be a total order.

Some of the terminology in the XML standards is confusing. For example, the *children* of a node n are defined to be all element or text nodes n' such that $(n, n') \in E$: thus, attribute nodes are not regarded as children although they do have parents! Also, and regrettably, there are a multitude of earlier models: that of the original specification [28], that of DOM [47], that of the so-called *XML information set* [29], and that of the earlier XPath [23].

Fortunately most important points about XML can be illustrated with the more simplified Definition 1, and we will use that throughout the chapter, pointing out whenever the additional details impact the logical foundations.

2.3 Some practical and principled issues

Memory Model To a programmer, a node is an object, which physically is a piece of allocated memory; in DOM, this object contains pointers so that both the parent, the children, and attribute namespace nodes can be reached in unit time. Thus, this model is fundamentally different from the storage model of S-expressions, recursive data types, and even many database models, where only children can be reached in unit time. In contrast, DOM programming is often based on extensive references to ancestors, often elegantly combined with XPath, a style of programming that is not built-in or even emphasized with recursive data structures. This difference has profound significance for XML programming concepts.

Document Order In practice, whenever the XML document is read from a file, the ordering $<$ is defined to be the *document order*—the order in which the nodes are listed in the file. When the XML document is generated otherwise, however, we must ensure that such a total order is given. For example, query languages like XQuery and XSLT must formally define the order in the XML outputs that they produce. In Fig. 2, the numbering of nodes we have chosen is in accordance with the document order.

Sequences of Elements Usually, the document node is required to have exactly one child, which must be an element. This node is sometimes called the *document element node*. However, it is very attractive to be able to work with sequences of trees; for example, if an XML file is used as a log of transactions, a file operation of append suffices to extend the log. The XML syntax already allows such representation in the form of files obscurely called "well-formed external parsed entities."

3 XML Schema Notations

For any XML application, there are two kinds of documents: those that make sense and those that do not. *XML schema languages* are notations for delineating the meaningful documents from those that are not, at least to a first approximation. It is claimed that the world needs only one schema language. In fact, we have quite a few, not only for historic reasons, but because there are several fundamental choices to make in schema language design.

3.1 DTDs

The most widely used such meta-language is called *DTD* (Document Type Description); it is part of the original definition of XML [28]. DTDs are very simple: for each element, a regular expression over element names specifies the allowed content. Figure 5 illustrates a DTD for the document in Fig. 1. For example, to specify that the content of the element `ingredient` is a sequence of `ingredients`, we may write (in the slightly arcane syntax of DTDs)

```
<!ELEMENT ingredients (ingredient*)>
```

```
<!ELEMENT recipe      (ingredients,procedure)>
<!ELEMENT ingredients (ingredient*)>
<!ELEMENT ingredient  (#PCDATA)>
<!ELEMENT procedure   ((#PCDATA|emph)*)>
<!ELEMENT emph        (#PCDATA) | emph>
<!ATTLIST recipe     name     CDATA #REQUIRED
                     estimated-time CDATA #REQUIRED
                     unit     CDATA #REQUIRED>
<!ATTLIST ingredient quantity CDATA #REQUIRED
                     unit     CDATA #REQUIRED
                     name     CDATA #REQUIRED>
```

Fig. 5. A DTD for the XML document in Fig. 1.

Such content, called *element content*, does not allow any text nodes (in particular, no white space nodes.) In general arbitrary regular expressions are allowed to define the content of an element. Regular expressions are formed from element names using concatenation (`,`), alternation (`|`), Kleene closure (`*` and `+`) and optional (`?`). The expression `ingredient*` above represents a regular expression with the Kleene closure operator. For another example consider the following definition of the content of `recipe`:

```
<!ELEMENT recipe ((ingredients|procedure)*,winelist?)>
```

This definition specifies that the `recipe` element may contain any sequence of `ingredients` and `procedure` subelements, intermixed in any order, followed optionally by a single `winelist` element.

Alternatively, an element may be specified to allow text, possibly interspersed with elements that have names in a specified set. For example, we may declare

```
<!ELEMENT ingredient (#PCDATA)>
```

The keyword `#PCDATA` stands for *parsed character data* and denotes the fact that the content of `ingredient` can be any sequence of characters[1]. The example below:

```
<!ELEMENT procedure (#PCDATA|emph)*>
```

says that `procedure` may have any text values interleaved with `emph` subelements.

Additionally, the allowed and required attributes of each element can be declared. For example,

```
<!ATTLIST ingredient name CDATA #REQUIRED>
```

[1] In Fig. 1 all `ingredient` elements have an empty content, i.e., an empty sequence of characters.

specifies that `name` is a required attribute of `ingredient`. Finally, and somewhat controversially, attribute declarations may be associated with defaults.

Usually, a document explicitly mentions the schema to which it is purported to conform. That allows the *schema processor*, the software that is often integrated with an XML parser, to read the document and process it with its associated schema. If the document satisfies the description, then it is called *valid* and the processor constructs the tree for the document, including inserted defaults. This notion of validated tree is often essential to optimization of further processing, say by a query processor, since the structural properties of the schema can be assumed.

Note that the DTD in Fig. 5 is recursive: an `emph` element may contain text interspersed with other `emph` elements.

Ignoring the attributes for a moment, there is a simple but elegant connection between DTDs and context free grammars. Namely each DTD corresponds to an *extended context-free grammar*, where productions may have regular expresions on their right hand side. Then, an XML document is valid with respect to the DTD precisely when its associated tree is a correct derivation tree for that grammar. We illustrate with the DTD in Fig. 5. The corresponding extended context free grammar is:

```
d           ::=  recipe
recipe      ::=  ingredients,procedure
ingredients ::=  ingredient*
ingredient  ::=  #PCDATA
procedure   ::=  (#PCDATA | emph)*
emph        ::=  #PCDATA
```

Here `d` is the start symbol in the grammar. Then the tree in Fig. 2 is indeed a derivation tree for this grammar, indicating that the XML document in Fig. 1 is valid with respect to this DTD.

3.2 Regular Expression Types

Extending recursive data types of functional programming languages, *regular expression types* allow us to enlarge the class of types of XML documents. We will illustrate here regular expression types as formulated in *XDuce* [48]. Fundamentally linked to tree automata for XML – see Sect. 5 – regular expression types are also adopted by XQuery and some schema languages. In the XDuce notation the DTD in Fig. 5 can be represented as in Fig. 6.

Here `Trcp, Tingrs, Tproc, ...` are *type identifiers* (or type names). Thus, `Trcp` is the type of a node whose tag is `recipe` and whose content consists of a node with type `Tingrs` followed by a node of type `Tproc`, and `Tingrs` is the type of a node with tag `ingredients` and with content consisting of a sequence of nodes of type `Tingr`. There are predefined atomic types, such as `String` and `Int`. An important idea of regular types is the separation between type identifiers, which act as nonterminals in a *tree grammar*, and XML tags, which form the labels of the tree.

```
Trcp    = recipe[Tingrs, Tproc]
Tingrs  = ingredients[Tingr*]
Tingr   = ingredient[String]
Tproc   = procedure[String | Te]
Te      = emph[String]
```

Fig. 6. An XML type in XDuce. The type corresponds to the DTD in Fig. 5.

3.3 XML Schema

XML Schema is an elaborate schema notation recently developed and approved by the W3C. The primer [34] is a readable introduction to the complex specification "XML Schema Part 1: Structures" [80] – that concern tree structure – and the less controversial "XML Schema Part 2: Datatypes" [10] – that deal with string types such as used in attribute values and in character data.

Like other recent schema proposals, XML Schema separates tags from types. Its type system is much more complicated than that of its competitors because of a number of apparently overlapping mechanisms for solving the problem of extending and reusing types. In particular, there is an object-oriented mechanism for derivation by extension, another one for redefinition, and a third one for substitution of elements. Additionally, a great number of concepts have been introduced to restrict the uses of these mechanisms; in this way XML Schema has become an extraordinarily complex notation.

Even so, the expressive power of XML Schema is less than simpler notations such as Xduce. For example, XML Schema stipulates that if some element has two children with the same tag then those children have the same type.

Fortunately, the core of XML Schema is rather straightforward. Given its growing significance, we illustrate its use in Fig. 7. We note that the syntax of XML Schema is itself XML. This is why the specification becomes voluminous although the only added information over the earlier DTD is the specification of the kinds of strings that may occur in the `estimated-time` and `unit` attributes of the `recipe` element. The schema uses namespaces in order to distinguish names that pertain to the meta language itself – those names that occur without a prefix – and names that pertain to the XML language being defined – those names that occur with a `r:` prefix.

Despite, or perhaps because of, the detailed and voluminous semantic explanations of the structural part of XML Schema [79], there is no formal semantics for the specification. In contrast, *RELAX NG* is easily described mathematically, indeed that is the approach to defining the language in [25]. Researchers have made some headway in interpreting XML Schema [75].

```
<schema xmlns="http://www.w3.org/2001/XMLSchema"
        xmlns:r="http://www.recipes-are-us.com/recipe-schema"
        targetNamespace="http://www.recipes-are-us.com/recipe-schema">

 <simpleType name="unitType">
   <restriction base="string">
     <enumeration value="h"/>
     <enumeration value="m"/>
   </restriction>
 </simpleType>

 <element name="recipe">
   <complexType>
     <sequence>
       <element ref="r:ingredients"/>
       <element ref="r:procedure"/>
     </sequence>
     <attribute name="name" type="string"/>
     <attribute name="estimated-time" type="float"/>
     <attribute name="unit" type="r:unitType"/>
   </complexType>
 </element>

 <element name="ingredients">
   <complexType>
     <sequence>
       <element ref="r:ingredient" maxOccurs="unbounded" minOccurs="0"/>
     </sequence>
   </complexType>
 </element>

 <element name="ingredient" type="string"/>

 <element name="procedure">
   <complexType mixed="true">
     <sequence>
       <element ref="r:emph" maxOccurs="unbounded" minOccurs="0"/>
     </sequence>
   </complexType>
 </element>

 <element name="emph">
   <complexType mixed="true">
     <sequence>
       <element ref="r:emph" maxOccurs="unbounded" minOccurs="0"/>
     </sequence>
   </complexType>
 </element>

</schema>
```

Fig. 7. An XML Schema for recipes

3.4 Schema Design Choices

There are several choices one can make in defining XML schemas, and the debate about which ones to make still rages. We discuss here a few of them along with some alternatives to XML Schema. For a detailed comparison of XML schema languages, see [54]

Regular expressions vs. Boolean logic Schema notations tend to describe attributes in less sophisticated ways than content (the sequence of children of a node). For whereas content descriptions rely on regular expressions, indicating potentially non-trivial relationships between elements, attribute descriptions allow no dependencies at all. This is a curious state of affairs, since there are often strong relationships among attributes themselves, and among attributes and content. In fact, even the XML Schema notation has many such dependencies – none of which can be described in the language itself. This is not a satisfactory state of affairs.

The problem has been addressed by RELAX NG [25], which generalizes regular expressions on elements to also include attributes. Regular expressions are easy to explain, probably easier than propositional logic, so this choice is an interesting philosophical and technical one. The DSD notation [53] tackles the problem by tying Boolean logic about attributes to regular expressions on content. Finally, it has been suggested to found schema notations on Boolean logic plus counting; in [13], the motivation is rooted in complexity considerations about the type checking of the plug operation (defined in Sect. 5).

Context dependencies DTDs associate a regular expression to each element name. Consequently, the allowed content of an element is the same, wherever in a document it is used. For example it is impossible to define a DTD where a `title` element under `employee` has a different structure from a `title` element under `book`. This is possible, however, if one separates tags from types as illustrated in the following example, using the XDuce notation for types:

```
Tr = things[(Te|Tb)*]
Te = employee(EmpTitle, Age)
Tb = book(BookTitle, Authors)
EmpTitle = title(First, Last)
BookTitle = title(String)
...
```

In these type definitions, `EmpTitle` and `BookTitle` are types with the same tag, namely `title`, but they have different contents: an `EmpTitle` contains a `First` subelement followed by a `Last` subelement, while a `BookTitle` contains a string. Furthermore, an `employee` element contains a `title` with the first kind of content, while a `book` element contains a `title` with the second kind of content. This cannot be expressed in a DTD, because there we need to specify the content of `title` regardless of the context where it occurs.

The essence of the this solution is to allow the type of an element to depend on the *context path* – its sequence of ancestors. But a philosophical and practical problem with the tree grammar approach is that it introduces a set of concepts, namely all the type names, outside of the vocabulary of the XML documents themselves. This is somewhat at odds with the self-describability of XML, since the node labels are no longer sufficient to describe the type of information. In fact, the types are implicit – computed from the grammar. Schema notations that completely avoid grammar concepts have been suggested, such as *Schematron* [49], a kind of constraint notation, and *Assertion Grammars* [71], which substitute types for expressions characterizing context paths.

Is a Type Ordained from Above or a Solution to Equations? The type equations of the previous example suggest that types are determined in a top-down manner and are a function of the context paths alone. Indeed, this is the view taken by most schema languages. But there is an alternative reading that allow equations to be nondeterministic characterizations of the whole tree: a document is typeable if and only if it is possible to find a type assignment to each node such that all the equations are satisfied. In this way, types acquire a semantics of tree automata on XML documents – which we explain in Sect. 5. Theoretically appealing, this semantics seem to be difficult to explain to practitioners. Only current one schema language, RELAX NG has taken this approach.

To Default or not to Default DTDs are not only descriptions of valid documents; they transform documents by inserting default and eliminating text nodes in a process known as *schema augmentation*. Most other schema languages are similar: they are both descriptive and transformational languages at the same time. Moreover, a schema in reality defines two XML languages: (a) the language of documents that through schema processing turn into augmented documents and (b) the language of such augmented documents. The argument for augmentation is that schema defaults are inherently linked to the usage of XML, so that defaults must be described along with the grammar. Ideally, a notation for XML documents would define (a) and (b) in terms of a logic or grammar notation that would also have an operational meaning of augmentation. No satisfactory solution to this problem has been found; for example, DSDs offer expressive, default insertion mechanisms inspired by [12], but their semantics are intricately tied to operational steps of parsing.

Closed World vs. Open World Assumption Schema languages mostly follow the tradition of programming languages: everything that is not explicitly specified to be allowed must not occur. This *closed world assumption* is fundamentally at odds with the extensibility assumption of XML. Schema languages offer technical escape hatches, such as allowing any content in certain cases (but that disallows any further properties of content to be specified) or to allow any content in foreign namespaces. Schematron relies on an open world assumption.

4 Programming and Querying XML

When it is internal to an application, XML data are often converted into DOM (see Sect. 2.2) and accessed via a standard API, in Java or C++. Not surprisingly, the graph-oriented DOM notation is not easy to work with. Additionally, XML data are sometimes external: it may be too large to fit in main memory (e.g. a database), it may be on a remote location, or it may be constructed on the fly, by some other application. So, there are important reasons for formulating specialized languages for exploring and querying XML. There are three important such languages defined by the W3C: XPath, XQuery, and XSLT.

4.1 XPath

The *XPath* language [23] defines expressions for navigating through an XML document, much like file path expressions for accessing Unix directories. XPath is intended to be a simple language, used as building block in more complex languages such as XQuery, XSLT, and XPointer.

We illustrate XPath through examples. Referring again to the XML data in Fig. 1, the XPath expresssion:

```
/recipe/procedure/em
```

returns the node n_{20} of Fig. 2, that is, the one represented by:

```
<em>vigorously</em>
```

Digging down a little deeper below the `em` node, we can select the text child n_{21}, using `text()`:

```
/recipe/procedure/em/text()
```

returns the string:

```
"vigorously"
```

In general, an expression returns a *sequence* of nodes: all nodes matching the expression, ordered according to document order. The XPath expression:

```
/recipe/ingredients/ingredient
```

returns the sequence n_6, n_{10}, n_{14} of nodes represented by:

```
<ingredient quantity="200" unit="g" name="flour"/>
<ingredient quantity="100" unit="g" name="sugar"/>
<ingredient quantity="2" unit="dl" name="milk"/>
```

To select attribute values, one prefixes attribute names with `@`, like in:

```
/recipe/ingredients/ingredient/@name
```

which returns:

```
"flour",  "sugar", "milk"
```

Thus XPath expressions may also evaluate to sequences of strings (as well as numbers; in fact, XPath offers various traditional value manipulations that fall outside the scope of this chapter).

An element or attribute reference, like `ingredient` or `@name` is called a node test. Other node tests are: `*` which matches any element node, `@*` which matches any attribute node, `text()` which matches any text node, and `node()` which matches any node. The `/` operator retrieves any child of its left operand and is called a *navigation step*. The other navigation step that we discuss here is `//` which retrieves any descendant. The descendant navigation step is a restricted form of a Kleene closure[2]. When no left operand is present for either `/` or `//` then it is assumed to be the root node, which we denoted d in Def. 1. For example the XPath expression:

```
//ingredient/description//time/*/text()
```

starts by following an arbitrariy chain from the root until it finds a node labeled `ingredient`, then fetches its `description` children, then finds their `time` descendants, then retrieves all their element children, and finally returns their text values.

So far we have illustrated only linear XPath expressions that unconditionally navigate down in the XML tree. *Filters*, sometimes called *predicates*, allow side conditions, making XPath expressions look like patterns. An example is:

```
//ingredient[manufacturer/address]/description/
                              time[@timezone]/text()
```

The conditions enclosed in `[...]` are the filters. The meaning of this whole expression is obtained in two steps. First, remove the filters to obtain a linear XPath expressions:

```
//ingredient/description/time/text()
```

[2] At the time of writing there is no general Kleene closure in the XPath standard.

This defines a set of nodes which in turn induces a set of paths in the XML tree. Second, retain only those paths for which the `ingredient` node satisfies the `[manufacturer/address]` condition (i.e., it has at least one `manufacturer` child, with a `address` child) and for which the `time` node satisfies the `[@timezone]` condition, i.e., it has such an attribute.

In practice filters are usually used to compare values, like in:

```
//ingredient[manufacturer/address/zip="12345"]/
         description/time[@timezone="EST"]/text()
```

The examples illustrated so far are absolute XPath expressions, in that they start at the root.

When used in other languages, XPath expressions may be relative, i.e., they may start at any node in the document, called the *context node*. The context node is specified by the application. In that case they are not prefixed with `/` or `//`. For example:

```
ingredient[@quantity="200"]/time
```

checks whether the context node is an `ingredient` node, and whether it has a `@quantity` attribute with value `200`, then returns its `time` child.

Figure 8 summarizes some of the features of the XPath language.

`ingredient`	matches an `ingredient` element
`*`	matches any element
`/`	matches the root
`/recipe`	matches a `recipe` element at the root
`recipe/ingredients`	matches a `ingredients` node having a `recipe` parent
`recipe//time`	matches a `time` node following a `recipe` at any depth
`//time`	matches a `time` at any depth
`time | place`	matches a `time` or a `place`
`@quantity`	matches a `quantity` attribute
`ingredient[@quantity="200"]/@unit`	the `@unit` attribute of an `ingredient` node with some `quantity` attribute equal to `200`

Fig. 8. Main features in XPath

Related Work In [31], a subclass of XPath expressions called *simple* is defined to shed light on decidability of containment and integrity constraints – the latter being logical statements in a restricted first-order logic. We mention more work on path constraints in Sect. 5.

4.2 XQuery

Large amounts of XML data stored on a data server is best accessed through an XML query language such as *XQuery* [11], currently being designed by the W3C. XQuery extends XPath with collection-oriented features: joins, union, and nested queries. The basic construct of XQuery is *for-let-where-return*, abbreviated FLWR and pronounced *flower*. Here is a simple example of a FLWR expression:

```
for $x in /recipe/instructions/instruction
where $x/@quantity > 50
return <specialIngredient> <quant> { $x/@quantity } </quant>
                           <name>  { $x/@name }     </name>
       </specialIngredient>
```

The query binds the variable `$x` to each `instruction` element, and for each binding where the `quantity` attribute is > 50 constructs a `special-Ingredient` element with two subelements, `quant` and `name`. For the XML data in Fig. 1, the result is:

```
<specialIngredient>
    <quant>200</quant>
    <name>flour</name>
</specialIngredient>
<specialIngredient>
    <quant>100</quant>
    <name>sugar</name>
</specialIngredient>
```

XQuery is primarily intended for database applications, and joins are an essential feature. The following query illustrates a join, by retrieving the ingredient's price, from a different part of the XML document:

```
for $x in /recipe/instructions/instruction,
    $y in /recipe/products/product
where $x/@name = $y/description/name/text()
return <cost> <name>  { $x/@name }        </name>
              <quant> { $x/@quantity }    </quant>
              <price> { $y/price/text() } </price>
       </cost>
```

Often the XML data is not stored natively as XML, but is constructed on-the-fly from other data sources, usually from a relational database. Then XQuery expressions are translated into another language, instead of being executed directly. For example, the query above may be translated into a SQL query like the following (the exact SQL expression depends, of course, on the schema of the underlying relational database):

```
select x.name, x.quantity, y.price
from   instruction x, product y
where  x.name = y.name
```

Related Work The question of using relational technology to store and process XML data is a very complex one. Several storage techniques have been proposed [73,40,30,83]. Query translation is discussed in [38,22,74,37]; a full translation of XQuery into SQL is described in [36]. One subtle issue that arises in representing XML data in relational databases is that the former has an ordered while the latter an unordered data model: this is addressed in [78]. A more recent overview of storage techniques can be found in [5].

4.3 XSLT

The initial goal of *XSLT* (*eXtensible Style Language Transformations*) [24] was to serve as a stylesheet language that converts XML to a formatting language, such as HTML, in order to be displayed by a browser. Since then, it has evolved into a rich language for transforming XML data to XML data. The main paradigm used in XSLT is that of a recursive function. With XSLT, the intuition is that of "walking the tree": one traverses the tree recursively and does some action at each node. Note that with XPath, one can also traverse the tree with `//`, but no action is done until the destination node is found. The following XSLT example illustrates the point. It copies an entire XML tree and replaces every `<em>` tag with a `<bf>` tag:

```
<xsl:template match="em">
    <bf> <xsl:value-of/> </bf>
</xsl:template>

<xsl:template match="*">
    <xsl:element>
        <xsl:apply-templates/>
    </xsl:element>
</xsl:template>
```

This XSLT program consists of two *template rules* (in general there can be arbitrarily many). Each template rule has a *match pattern* and a *template*. Consider the first rule. The match pattern is `em`: this is an XPath expressions matching an `<em>` tag (in general, it can be any XPath expression). The template is:

```
<bf> <xsl:value-of/> </bf>
```

Here `<bf>` and `</bf>` are programmer defined tags, while the element `<xsl:value-of/>` stands for an XSLT function. In general, the template consists of an arbitrary XML fragment interleaving output elements with XSLT functions. The meaning is the following. Whenever an input node matches `em`, an output of the form `<bf>...</bf>` is constructed. We examine now the content between the output tags: `xsl:value-of` is a predefined function in XSLT returning the text value of the current node. In the output tree, this text is made into a text node, which becomes the content of the `bf` element.

Consider now the second rule. Here the match pattern is `*`, matching any XML element, while the template is:

```
<xsl:element> <xsl:apply-templates/> </xsl:element>
```

The function `xsl:element` constructs a new element with the same tag as the current node in the input XML document. The XSLT function `xsl:apply-templates` calls the entire XSLT program recursively on each child of the current node. In effect, this rule alone copies the entire XML tree, except for attribute and text nodes: it copies the current element node, then calls the program recursively on the element children, copying them too.

Note that there is a conflict between the two rules: an element named `em` matches both templates. XSLT has a rather arcane set of rules for resolving such conflicts. The main idea is that the most specific rule will be selected: in our case the first rule.

The program in the example above acts like one recursive function that is called on each node in the tree. In XSLT we can simulate multiple, mutually recursive functions, using *modes*. A mode is a name given to a template rule, and all template rules having the same mode form one function. The example below illustrates this with a program that changes `em` to `bf`, but only directly under `instruction`, and that leaves all other `em` unchanged.

```
<xsl:template match="em" mode="f">
    <bf> <xsl:value-of/> </bf>
</xsl:template>

<xsl:template match="*" mode="f">
    <xsl:element> <xsl:apply-templates/> </xsl:element>
</xsl:template>

<xsl:template match="instruction">
    <xsl:element> <xsl:apply-templates mode="f"/> </xsl:element>
</xsl:template>

<xsl:template match="*">
    <xsl:element> <xsl:apply-templates/> </xsl:element>
</xsl:template>
```

As before the document is copied recursively, starting from the root. When an `instruction` is reached, then the copying continues, but in the `f` mode. The `em` tag is changed to the `bf` tag only in the `f` mode.

Related Work The reference [82] provides a formal model of XPath and XSLT pattern matching, whereas [9,58] discuss the meaning and expressive power of XSLT from a complexity theoretic point of view.

5 The Theory of Regular Tree Languages

Central to both XML schemas and query notations, regular tree languages are generalizations of elementary concepts in computer science. We review these languages here emphasizing their automata-theoretic origin and their connection to monadic second order logic.

Regular tree languages are defined over trees that have labels but no values, i.e., they are subsets of $\mathcal{T}_\Sigma$ rather than $\mathcal{T}_{\Sigma,D}$. Recall from Sec. 2.2 that we have a canonical "forgetful" function $\mathcal{T}_{\Sigma,D} \to \mathcal{T}_\Sigma$, hence any subset of $\mathcal{T}_\Sigma$ implicitly defines a subset of $\mathcal{T}_{\Sigma,D}$, namely, its preimage under the forgetful function.

Our definitions proceed from the case of ranked trees, where each label is associated with a fixed number of children.

Definition 2. Let Σ be a finite, ranked alphabet: that is, Σ is partitioned into disjoint sets, $\Sigma = \Sigma_0 \cup \Sigma_1 \cup \ldots \cup \Sigma_k$, where the symbols in Σ_r are said to have rank r, for $r = 0, \ldots, k$. A *tree automaton* over Σ is $A = (\Sigma, Q, F, \delta)$ where Q is a finite set of states, $F \subseteq Q$ is the set of final states and $\delta \subseteq \bigcup_{i=0,k} Q \times \Sigma_i \times Q^i$.

Given a tree automaton A and a tree $t \in \mathcal{T}_\Sigma$, a computation proceeds bottom-up, by assigning a state to each tree node. Whenever a node labeled $a \in \Sigma_i$ has i children, for some $i = 0, \ldots, k$, which have been assigned states $q_1, \ldots, q_i$, then we can assign state q to that node if $(q, a, q_1, \ldots, q_i) \in \delta$. Note that this definition also covers the case of a leaf node, which is labeled with an $a \in \Sigma_0$. This assignment is a nondeterministic process, i.e., at each step we may choose between several states q that can be assigned. *A accepts t* if we can assign a final state $q \in F$ to the root node. A *regular tree language* is a subset of $\mathcal{T}_\Sigma$ that is accepted by some tree automaton A.

A technical issue that arises in adopting the notion of regular tree languages to XML is that the XML trees are unranked (see Def. 1) while Def. 2 requires that the alphabet Σ be ranked.

There are two ways to circumvent that XML trees are unranked, and they turn out to be equivalent. The first way is to encode unranked trees as ranked binary trees [70]: an unranked regular tree language is then defined to be the preimage of a (ranked) regular tree language under this encoding. There are several natural ways to encode unranked trees as ranked trees, but the definition is robust with respect to the particular choice of the encoding. The second way is to generalize tree automata to unranked trees. To do so, note that δ of Def. 2 can be viewed as a function $\delta : Q \times \Sigma \to 2^{Q^*}$. Now, given an unranked alphabet Σ, define a tree automaton for unranked if trees to be $A = (\Sigma, Q, F, \delta)$ where Q, F are as in Definition 2 while $\delta : Q \times \Sigma \to 2^{Q^*}$ is such that for any q and a, the set $\delta(q, a)$ is a regular language over the alphabet Q [15]. The state assignment again proceeds bottom-up: whenever a node labeled $a \in \Sigma$ has k children that have been assigned states $\overline{q} = q_1, \ldots, q_k$, we may assign state q to that node provided that $\overline{q} \in \delta(q, a)$. The automaton accepts the tree if we can assign a state $q \in F$ to the root node. An unranked regular tree language is defined to be a set of unranked trees accepted by an unranked tree automaton.

As in the case of ranked automata it is possible to convert every tree automaton into a *deterministic* tree automaton, i.e., where the following property holds: for all $\overline{q}$ and a there is a unique q such that $\overline{q} \in \delta(q, a)$. It can be seen that the usual subset construction can be generalized to trees such that this property can be assumed to hold.

The Logic-automaton Connection An alternative way to define properties on trees, including regular tree languages, is based on logic. For that purpose, we view a tree $t \in \mathcal{T}_\Sigma$ as a finite structure, which is exposed through unary predicates $a(x)$, for $a \in \Sigma$, where $a(x)$ holds if and only if $\lambda(x) = a$, a binary document order predicate $<$, and a binary $\texttt{child}(x, y)$ predicate, which holds if and only if y is a child of x. The *first order (FO) logic* on trees allows us to express first order formulas over such structures by quantifying over nodes in the tree. The *monadic-second-order (MSO) logic* of such structures adds second-order variables interpreted as sets of nodes, along with quantification over these variables. Given a formula φ in FO or in MSO and a tree $t \in \mathcal{T}_\Sigma$, we denote $t \models \varphi$ if the formula φ is true in the structure corresponding to t. The following is an adaptation of a classic result on the connection between regular languages and monadic second order logic, see [27]:

Theorem 1. *Let Σ be a finite (unranked) alphabet. Then a set $L \subseteq \mathcal{T}_\Sigma$ is a regular tree language if and only if there exists an MSO formula φ such that $L = \{t \mid t \models \varphi\}$.*

As a consequence, the MSO logic of XML trees is decidable: to test if a formula φ is satisfiable in some model one first constructs the tree automaton A that accepts the set $\{t \mid t \models \varphi\}$ then tests if A accepts at least some tree (by a kind of accessibility test).

The proof of the Theorem shows that tree automata can represent the set of satisfying interpretations for any subformula. The basic idea is to augment the alphabet with binary components, one for each second-order variable. (First-order variables are treated as singleton second-order variables.) In this way, an extended label can record the membership status – true or false – for each variable at a position. Thus, a labeled tree now represents both the original labels and an interpretation of the second-order variables. Then, tree automata that recognize the sets of satisfying interpretation can be constructed using a simple induction on formulas. For example, conjunction corresponds to product of automata and the existential quantification corresponds to the construction of a nondeterministic automaton, followed by determinization.

The Plug Operation and Minimal Automata Finally, if we define a *pointed tree* ρ to be a tree over Σ except that one leaf is labeled $\bullet$, then we may compose ρ with any tree τ by replacing the leaf labeled $\bullet$ with τ. This *plug operation* defines a tree denoted $\rho \cdot \tau$. It is now not hard to show that the relation $\approx_L$ defined by $\tau \approx_L \tau' \Leftrightarrow (\forall \rho : \rho \cdot \tau \in L \Leftrightarrow \rho \cdot \tau' \in L)$ is an equivalence relation of finite index. In fact, it is a congruence relation: if $\tau_i \approx_L \tau_i'$, $1 \leq i \leq k$, then $a(\tau_1, \ldots, \tau_k) \approx_L a(\tau_1', \ldots, \tau_k')$, where $a(\tau_1, \ldots, \tau_k)$ here means the tree that has a root labeled a with k children being the roots of the τ_is. Moreover, it is possible for each $a \in \Sigma$ to define in a natural way an equivalence relation on sequences of equivalence classes corresponding to the transition relation. The details of how constructions lead to minimal automata are provided in [15].

Uses of Tree Automata With tree languages essentially behaving as nicely as conventional regular languages, it is not surprising that they have found use in the study of program analysis, see e.g. [50]. Recently, it is also shown to be feasible to use tree automata calculations according to the logic-automaton connection in practice [51]. Tree grammars are increasingly being used in implementations of program analysis [55] and for type inference in programming languages that allow a plug operation [13]. Thus, the investigation of tree automata and grammars holds considerable promise.

Regular Tree Languages and XML types A regular tree language $L \subseteq \mathcal{T}_\Sigma$ can be thought of as an XML type. Recall from Sect. 3 that an XDuce type consists of a set of type definitions: each definition has a type variable on the left hand side, and a tag followed by a regular expression on type variables on the right hand side. Since $\delta(q, a)$ in the definition of the tree automaton is a regular language, it is not hard to see that the XDuce content specifications are nothing but a syntax for regular expressions defining such languages. In other words, XDuce types and regular tree languages are equivalent. As a consequence every DTD, and every XSchema defines a regular tree language, but it can be shown that both these formalisms are strictly less expressive than regular tree languages: for example, neither can specify the set of all trees that contains at least one node – anywhere in the tree – labeled a. The reason is that the type system of these notations are entirely controlled by context paths, see Sect. 3.4. An exception is RELAX NG, which expresses exactly the class of regular tree languages.

Related Work More topics about automata, logic, and XML are discussed in the survey [35]. A general introduction to tree automata can be found in [27].

6 XML Query Languages vs. Automata and Logic

Languages for querying and transforming XML trees occur critically in applications such as XML data exchange and native XML data processing. In data exchange we often need to transform an XML document conforming to one schema into another document, representing the same information but represented according to a different schema. This is needed, for example, when XML data crosses organization boundaries. The core of a native XML engine consists of a query processor, which needs to optimize and execute queries over the XML data, much like a relational database query processor. We discuss here some of the theoretical investigations into the foundations of XML querying and transformation formalisms.

In Sect. 5 we already saw the close correspondence between types of XML documents on the one side and automata and logic on the other side. Besides this match there are two other well-known striking connections between query mechanisms and logics that seem relevant in the context of XML querying:

- The core of the relational database query language SQL has the same expressive power as first-order logic. This fact has been intensively exploited in the theory of relational databases.
- Regular expressions, which are frequently used for path navigation, e.g., in XPath, XQuery and XSLT, have the same expressive power as monadic-second-order logic.

In this section, we consider the relationship between XML query and transformation languages and corresponding logics and automata.

Why are we interested in this relationship? There are two main reasons: we want to understand the expressive power of query languages and we want to find methods for efficient query evaluation. The characterization of a query language by a logic is the starting point for an understanding of its expressive power. It facilitates the comparison with other languages, indicates possible extensions and makes it possible to find out which queries can not be expressed. On the other hand, an equivalent automaton model gives rise to efficient query evaluation (usually linear time in the size of the document) and to static analysis of queries, e.g., whether a query ever produces a result or whether the result set of one query is always contained in the result set of an other query.

In practice, we never get an exact correspondence between an XML language and a logic or an automata model. As in the case of SQL, the best we can hope for is a match between the core of a language or between certain features and corresponding logics and automata.

Research in this area can be roughly divided into two categories. One focusses on existing features in XML languages, especially on regular path expressions and studies them on trees as well as on graphs. We describe some of this work in Sect. 6.1.

The other category considers the more powerful MSO logic but restricts attention to queries that refer only to the tree structure of XML documents, ignoring links and comparisons between data values in different parts of a tree. Results of this approach are presented in Sect. 6.2.

Sect. 6.3 presents in more detail results on a specific, important question: the typechecking problem for XSLT transformations. The proofs of these results make use of the correspondence between XML types and MSO logic as well as of a characterization of XSLT transformations by a certain kind of automata and in turn the description of their behaviour by MSO formulas.

6.1 Query formalisms based on regular path navigation

We first describe several kinds of query formalisms which are based on regular path expressions and their extensions. Then we consider corresponding automata and logics.

Regular path expressions A *regular path expression* R is simply a regular expression over labels. A path p from a node x to a node y *matches* a path expression R if the sequence of labels on p is a string in the language over labels defined by R; we also say that R *holds* on the path. Hence, R induces a binary predicate, also denoted R, on nodes, namely the set of pairs (x, y) for which R holds on the path from x to y. For example, the path expression

$$x\ \texttt{chapter.(section|table)}^*\texttt{.figure}\ y$$

defines a binary predicate that holds of any pair of nodes (x, y) that are connected by a path whose labels spell a word in the regular language `chapter.(section|table)`*`.figure`. Sometimes one is interested in the case where x is fixed to be the root; R then simply defines a set of nodes.

For most queries, the full power of regular expressions is not needed. Some languages, e.g., the core of XPath as exemplified in Sect. 4, use only a fragment of regular expressions: concatenation, union, a wildcard for a single node with an arbitrary label, and a wildcard for an arbitrary sequence of nodes with arbitrary labels. The latter two features are replacements for the more powerful Kleene-star operator of regular expressions. Expressions without this operator (but with negation instead) are called *star-free* and have an interesting logical characterization. They exactly correspond to first-order logic [59]. Through appropriate use, negation can actually express path conditions that seem to require Kleene star at first sight, like the displayed expression of the book example above. Nevertheless, this sophisticated use of negation is not suitable for a user-friendly XML query language. Therefore, it is an interesting question, whether there is a simpler kind of specification of star-free conditions or whether there is another robust class of path conditions below the first-order expressible that have a simple syntax.

The path expressions we have seen so far can be either used to define a set of nodes, if the first node is assumed to be the root, or to define a set of pairs of nodes. In general, one might be interested to specify relations of arbitrary arity. This can be easily obtained by combining path expressions to form *conjunctive queries*. As an example, the query

$$\begin{aligned} q(x,y,z) = {} & (root\ \texttt{recipe} \cdot \texttt{ingredients} \cdot \texttt{ingredient}\ w)\ AND \\ & (w\ \texttt{manufacturer} \cdot \texttt{name}\ x)\ AND \\ & (w\ \texttt{@quantity}\ z)\ AND \\ & (w\ \texttt{description} \cdot \texttt{name}\ y) \end{aligned}$$

selects all triples (x, y, z), where x is the name of the manufacturer of ingredient y from which the quantity z is used [41,19].

A similar kind of tree-structured specification of patterns is possible with filters in XPath expressions (cf. Sect. 4), although in this case the result is only a set of nodes.

Regular path expressions on general graphs have been studied in a series of papers [17,21,18,20]. This work considers query evaluation and query rewriting in the presence of views. In *view-based query evaluation* a target query q is given by a regular path expression or a conjunction of such expressions and the answers to other queries $q_1, \ldots, q_k$, called *views*, are assumed available. The goal is to compute an (as good as possible) answer to q by using these views. In *query rewriting* the views are not given. Instead, the goal is to find an expression which uses $q_1, \ldots, q_k$ and always gives a good approximation for q. The cited papers investigate the complexity of these problems. In some cases the algorithms are based on automata.

For tree-structured data, the path expressions we have just seen navigate only along vertical paths that are oriented from the root to the leaves. It also makes sense to consider *horizontal regular expressions* that allow navigation along the children of a node. Of course, we we already know from Sect. 3 that DTDs specify the allowed content of a node as a regular expression. This kind of horizontal navigation is also possible in XPath along so-called sibling axes.

An alternative to the simultaneous imposition of separate regular expressions is to use a single expression over an extended vocabulary of navigational commands. *Routing expressions* [52] or the similar *caterpillar expressions* [14] provide such commands for moving up, testing the label, and selecting a child by moving down, left, or right. As an example the expression $\texttt{up} \cdot \texttt{ingredients} \cdot \texttt{ingredient}$ describes, for each procedure element of a recipe document the paths to the corresponding ingredients. The document order relation can be formulated as a caterpillar expression, but cannot be formulated as a conjunctive query.

Automata for path expressions Caterpillar expressions have an automata-theoretic formulation as a *tree-walk automaton*, which is a finite-state machine that visits one node of an input tree at a time. Upon reading the label of the node, the automaton may nondeterministically – but according to its transition table – decide to move up, move down to the first child, move left (to a sibling), move right. Instead of moving, the automaton may test the currently visited node for whether it is the first or last child or whether it has children. Each such transition is accompanied by a state change according to the transition table. By designating initial and final states, tree-walk automata represent binary relations, the same set as those defined by caterpillar expressions.

A tree-walk automaton also represents a tree language: the set of trees for which the automaton, started at the root, reaches an accepting state somewhere. It is not difficult to see that deterministic as well as nondeterministic tree-walk automata can be emulated by standard tree automata. In the other direction, it would perhaps seem counter-intuitive if any tree automaton could be emulated by a tree-walk automaton. Surprisingly, the status of this problem is open. It is even unknown whether all regular tree languages can be captured by deterministic tree-walk automata. As the power of tree-walk automata seems to be relatively weak, several extensions have been studied, e.g., with more than one head. One particular extended kind of tree-walk automata, so-called pebble automata, is discussed in Sect. 6.3 below. Tree-walk automata are further studied in [32,33,66].

Logics and path expressions Since star-free path expressions can be expressed in first-order logic, which is closed under conjunction, they also indirectly capture conjunctive queries over such expressions. But FO logic seems to be the more serious candidate for being the logical foundation of XML querying. We next mention some other logics that have been considered in the literature.

In [7] several existential fragments of first-order logic are studied which correspond to certain parts of the XPath language.

In [65] first-order logic is augmented by both vertical and horizontal regular expressions over formulas. E.g., an expression $\varphi\,\psi^*$ is true at a node if its first child fulfils a formula φ and each of the remaining children fulfil ψ.

Although it remains open whether tree-walk automata are equivalent to MSO logic, their expressive power can be characterized by natural logics. The probably somewhat stronger model of pebble automata exactly corresponds to FO logic augmented by the unary transitive closure operator (see [69] for a reference). The latter expresses whether a pair of elements is in the transitive closure of a (defined) binary relation. Deterministic and nondeterministic tree-walk automata can be captured by formulas which consist of one (deterministic or nondeterministic) transitive closure operator in front of an FO formula (and a built-in depth predicate) [66].

6.2 Query formalisms with the expressive power of monadic-second-order logic

Next, we study formalisms that are equivalent to MSO logics. Recall that equivalence between tree automata and MSO logic in Sect. 5 applies to recognition of sets of trees, not to queries. It is a pleasing feature of logics, however, that they can be easily adapted to express queries of arbitrary arity: to get a k-ary query simply write a formula with k free first-order variables. We will see soon, that the analogous step for automata is much more involved.

In this subsection, we first describe automata models, then we have a look at other kinds of formalisms. Afterwards the complexity of evaluating queries that have MSO power is considered.

Automata for MSO queries We saw in Sect. 5 how the definition of classical tree automata can be extended to handle the unranked trees representing XML documents. It is straightforward to amend *nondetermistic* bottom-up or top-down tree automata so that they compute unary queries; it suffices to view the set of accepting states as those that select the nodes to be included in the answer set. To get a *deterministic* model with MSO power, the requirement that the automaton makes a single pass over the tree must be dropped. In fact, the decision as to whether a node is selected may depend on parts of the tree both below and above the node under consideration.

In [15], two-way deterministic tree automata were defined. At each point of the computation the heads of such an automaton are placed along a *cut* of the tree, i.e., a set of nodes that intersects each path from a leaf to the root exactly once. In one step the automaton can move either from a node v to its children or, if all children of v carry a head, up to v. The new states are defined by means of regular languages as explained in Sect. 5. In [64], such automata were extended by node selecting states to express unary queries. It turns out that the resulting model is too weak to capture all MSO definable queries. The obstacle is that not enough information information can be passed among siblings. For instance, the simple query "select each `ingredient` node that is the first such child for some parent" cannot be expressed. To bridge the gap, *query automata* must be accessorized with *stay-transitions*. During such a transition, a two-way string automaton with output transforms a sequence of states of the children of a node v into a new sequence of states. Unreined, this feature makes query automata too powerful (e.g., equivalent to linear space bounded Turing machines on trees of depth 1). Therefore such transitions are allowed only to be used a constant number of times per node v. Defined in this fashion, query automata capture precisely the unary queries expressible in MSO logic.

A different automata-based approach is taken by [63]. There a deterministic sequential automaton that traverses the tree twice, in document-order and reverse document-order, is equipped with a pushdown store. Again this model has the full expressive power of MSO logic.

Other formalisms to express MSO queries Automata are not the only operational models that have the same expressive power as MSO logic. We mention two other approaches with the same expressive power. In [68], it is shown how Boolean-attributed grammars over context-free grammars – extended with regular expression operations as in DTDs – constitute a query language. The other MSO-equivalent notation is that of monadic Datalog programs [45]. These programs may be further restricted to a fragment that is in an intuitive correspondence with a visual notation – while still being MSO-equivalent. This fragment is used in the Lixto project [56] to represent extraction rules for Web pages.

Complexity issues The *data complexity* of a query is the amount of resources needed to evaluate it as a function of the size of the input data. For most query languages, and indeed even for MSO logic with free first-order variables, the data complexity is linear time. Such efficient evaluation can be obtained from each of the operational models discussed above. In [42], it is even shown that MSO queries with free second-order variables can be evaluated in linear time data complexity (in the size of the structure and of the output) on structures of bounded tree-width; in this case, the output is a set of structures.

In all these results, however, the query is considered fixed and only the XML document is viewed as input. This point of view is partly justified by the assumption that the query itself is much smaller than the XML data. In practice, it is often relevant to consider the query itself as part of the input. This is the point of view of *combined complexity*, which better reflects the actual query mechanism. Fortunately, for the operational models, the complexity remains linear in both the query size and the size of the input. But the picture changes drastically when the query is given as an MSO formula. The obvious approach, to translate the formula into an automaton, yields a non-elementary algorithm, see [27]. (A *non-elementary* function is one that cannot be bounded by a fixed stack of exponentials.) In fact, each quantifier alternation may increase the running time of such an algorithm by an exponential.

In [43], it is shown that there is essentially no better way to evaluate MSO formulas, if the time complexity has to be polynomial in the size of the document. More precisely, unless **P=NP**, if an algorithm evaluates MSO formulas φ on documents t in time $O(f(|\varphi|)p(|t|))$ with p a polynomial, then f is non-elementary. The non-elementary complexity can be brought down to exponential through syntactic restrictions on formulas without losing any expressive power (but the formulas may have to be non-elementary bigger) [65].

On the other hand, a straightforward evaluation strategy based on structural induction (and without using automata) also has exponential time complexity, but works in polynomial space. Actually, the combined complexity problem is complete for **PSPACE**.

6.3 The XML Typechecking Problem

Recall that an XML *type* is a regular tree language $\tau \subseteq \mathcal{T}_\Sigma$. The *XML validation problem* is: we are given a tree $t \in \mathcal{T}_\Sigma$ and a type τ and we have to decide whether $t \in \tau$. For example, the tree in Fig. 2 is valid with respect to the type `recipe` of Sect. 3.1. In the *typechecking problem* we are given a program P, defining a function $P : \mathcal{T}_{\Sigma,D} \to \mathcal{T}_{\Sigma',D}$, two types $\tau \subseteq \mathcal{T}_\Sigma, \tau' \subseteq \mathcal{T}_{\Sigma'}$, and need to decide whether $\forall x \in \tau$, $P(x) \in \tau'$. Here and in the sequel the notation $x \in \tau$ means that, after erasing the data values from x the resulting tree is in τ; similarly for $P(x) \in \tau'$.

An XML typechecker is needed in virtually any application where XML documents are generated automatically, whether by a program or by a query. One should think of the typechecker as a module which, upon analyzing the program, the input type, and the output type, decides whether all documents that can be produced by the program are valid, and returns `yes` or `no` accordingly. If the answer is `no`, then we would also like to know where in the program typechecking failed. This however may be hard, because typechecking is a global property and it may be impossible to say which subexpression caused the typechecker to fail.

A related problem is the *type inference problem*. Here, we are asked to compute, for the given program P, the type $P(\tau) = \{P(x) \mid x \in \tau\}$. When type inference is possible, we have a simple solution to the typechecking problem: given P, τ, τ', first compute $P(\tau)$, then check if $P(\tau) \subseteq \tau'$. Early work on typechecking document transformations [62], XDuce [48] and XQuery approach typechecking through type inference. However, type inference is usually impossible, because the set $P(\tau)$ is not a regular tree language. For example, consider the XQuery program P below:

```
<result>
    for $ in /doc/elm
    return <a/>
    for $ in /doc/elm
    return <b/>
    for $ in /doc/elm
    return <c/>
</result>
```

On an input like:

```
<doc> <elm/> <elm/> ... <elm/> </doc>
```

with n occurrences of `<elm>`, it returns an output of the form:

```
<result>
    <a/> ... <a/> <b/> ... <b/> <c/> ... <c/>
</result>
```

with n a's, n b's, and n c's. We say, in short, that on the input $\texttt{elm}^n$ it returns the output $\texttt{a}^n.\texttt{b}^n.\texttt{c}^n$. Then, for a given input type τ that allows arbitrarily large numbers of elm elements the output, $P(\tau)$ is not a regular language, not even a context free one. Systems in practice usually approximate the inferred output type to a set that is larger than $P(\tau)$. XQuery's standard type inference system would approximate the output type to $\texttt{a}^*.\texttt{b}^*.\texttt{c}^*$. Typechecking can still be attempted, but it may result in false negatives, i.e., to some correct programs being rejected by the typechecker. For example, the reader may check that the program above correctly typechecks with respect to the input type:

```
<!ELEMENT doc (elm*)>
```

and the output type:

```
<!ELEMENT result ((a,a*,b,b*,c,c*)?)>
```

but the XQuery standard typechecker will fail to typecheck it, because $\texttt{a}^*.\texttt{b}^*.\texttt{c}^* \not\subseteq (\epsilon \mid (\texttt{a}.\texttt{a}^*.\texttt{b}.\texttt{b}^*.\texttt{c}.\texttt{c}^*))$. The only solution to such false negatives is to ask the programmers to rewrite their programs in order to help the typechecker – a rather serious annoyance in practice.

Another approach to typechecking is via *inverse type inference*. Here, we are asked to compute, for a given program P and output type τ', the type $P^{-1}(\tau') = \{x \mid P(x) \in \tau'\}$. As before, when inverse type inference is possible then typechecking can be done as follows: given P, τ, τ', first compute $P^{-1}(\tau')$, then check if $\tau \subseteq P^{-1}(\tau')$. Surprisingly, inverse type inference is possible, for a large class of XML transformations discussed below. For example, considering the program above, assume the output type to be: $\tau' = \{\texttt{a}^{2m}.\texttt{b}^{3n}.\texttt{c}^{5k} \mid m \geq 0, n \geq 0, k \geq 0\}$. This is indeed a regular language, expressible as $(\texttt{a}.\texttt{a})^*.(\texttt{b}.\texttt{b}.\texttt{b})^*.(\texttt{c}.\texttt{c}.\texttt{c}.\texttt{c}.\texttt{c})^*$. Consider now the inverse type inference problem for the program P above: the reader may verify that $P^{-1}(\tau') = \{\texttt{elm}^{30p} \mid p \geq 0\}$, which is a regular language ($30 = 2 \times 3 \times 5$).

XML Transformations The class of XML transformations for which we discuss inverse type inference and typechecking are k-pebble transducers [61]. Like tree-walk automata these devices walk over the input tree in any direction (up/down-left/down-right). There are two extensions to the tree walk automata.

First, there are up to k heads (called *pebbles*) that are placed on the input tree at any given time, and that can move independently. The number $k \geq 1$ is a parameter of the transducer. The following restriction is imposed on the pebbles' movement: at any given time, only the highest ranked pebble can be moved. To move a lower ranked pebble, the higher ranked pebble must first be removed from the tree. This restriction still allows us to express nested loops in a k pebble tree transducer, like:

```
for x in nodes(t) do
   for y in nodes(t) do
       ...
```

Here pebble x is the lower ranked, and when it advances the higher ranked pebble y is taken from the tree (and, later, placed on the root). To see an example of an operation that is ruled out by the restriction, consider an XML document which is a list of a's followed by a list of b's, i.e., $\mathtt{a}^m.\mathtt{b}^n$. Assume we want to check whether $m = n$: we could do this with two pebbles iterating in sync over the a, and over the b's respectively. This would allow us to recognize the language $\mathtt{a}^n.\mathtt{b}^n$, which is not a regular language: the restriction, however, prohibits us from doing so.

Second, k-pebble tree transducers produce an output tree, rather than just accepting or rejecting a tree. For that they allow a new kind of transition called an *output transition*. Here an output node is produced in the output tree, and a fresh copy of the k-pebble transducer is created for each child of that node. These copies run in parallel, without communication, and compute the subtrees of the output node. They all start in the same configuration inherited from the original transducer (i.e., the same placement of the pebbles) except for the initial state, which depends on the rank of the child. To see a simple example, the following 1-pebble transducer copies a binary input tree to the output tree. It has three states, q_0, q_1, q_2. In state q_0 it produces an output node, with the same label as the current input node, and with two children: the transducers for those children are initialized in states q_1 and q_2 respectively. In state q_1 it moves the current pebble to the left child and enters state q_0. In state q_2 it moves the current pebble to the right child and enters state q_0.

k-pebble tree transducers can express an important fragment of XSLT that includes recursive traversal of the input tree and the use of variables and parameters. However, it cannot express *joins*, i.e., the data values of two parameters cannot be compared for equality.

In summary, k-pebble transducers can express transformations consisting of one or several recursive traversals of the input tree, like in XSLT, but cannot express joins, like in XQuery. Then the following holds:

Theorem 2. *[61] For any k-pebble tree transducer P and any regular tree language τ, $P^{-1}(\tau)$ is also a regular tree language. In consequence the type-checking for k-pebble tree transducers is decidable: given P, τ, τ', it is possible to decide whether $\forall x \in \tau, P(x) \in \tau'$.*

The algorithm resulting from the proof in [61] is very inefficient (hyper-exponential). A more efficient algorithm (exponential) for a certain fragment of XSLT is described in [81].

The restrictions on the k-pebble transducer cannot be lifted without compromising Theorem 2. If one allows all pebbles placed on the tree to move in parallel, then one can check if the input is of the form $\mathtt{a}^n.\mathtt{b}^n$, by moving one pebble over the `a`'s synchronously with another pebble over the `b`'s: this suffices to construct a transducer P for which $P^{-1}(\tau')$ is not regular.

More important for practical applications is the fact that one cannot lift the restriction on data values. We explain this in more detail. Recall that XQuery can express joins by checking equality conditions between data values. It is easy to extend k-pebble transducers in the same way: they are now allowed to check if $\nu(x) = \nu(y)$, for two nodes x and y pointed to by two different pebbles. Call such a machine a k-pebble transducer with data values. Now the k-pebble transducer really defines a function $\mathcal{T}_{\Sigma,D} \to \mathcal{T}_{\Sigma',D}$, in that the shape of the output tree and its labels depend on the data values in the input tree. To see such an example, consider the function which copies the input tree if all data values occurring in the tree are distinct, and returns an empty tree otherwise. Unfortunately, typechecking becomes undecidable in this case:

Theorem 3. *[61,77] The following problem is undecidable. Given a k-pebble tree transducer with data values, P, and two regular tree languages τ, τ', decide whether P typechecks w.r.t. τ and τ': $\forall x \in \tau . P(x) \in \tau'$.*

This in itself does not answer the question whether typechecking is decidable for the non-recursive fragment of XQuery, consisting of (possibly nested) FLWR expressions, because k-pebble transducers allow complex traversal of the input tree, that is not expressible with simple FLWR expressions. This question is settled in [3] which essentially showed that typechecking is here undecidable. It does become decidable, however, if one further imposes restrictions on the query language or the output type.

7 Conclusion

We have described XML formalisms that have logic foundations and that are emerging as cornerstones of new Internet-driven technology: schemas, query languages, and type checking.

This chapter has had to omit many more advanced developments. For example, XML under constraints is an area that span formalisms for specifying XML keys [16], the satisfiability problem for a set of constraints [35], and constraints for the purpose of storing and querying XML data that is redundantly stored and indexed in relational format [2]. Other research has addressed the XPath query containment problem [67,60], structural properties of XPath [7], the XML tree encoding problem (how to efficiently represent relations among nodes) [1,4,26], the XML normalization problem [6], XML stream processing [72], alternative definitions of regular tree languages [57,8], and monadic queries over XML [46].

Thanks to the explosion of XML-related research, these references are just to be construed as a sampling of current directions. Much work remains to be done.

References

1. S. Abiteboul, H. Kaplan, and T. Milo. Compact labeling schemes for ancestor queries. In *Proceedings of ACM-SIAM Symposium on Discrete Algorithms*, 2001.
2. Alin Deutsch and Val Tannen. Reformulation of XML Queries and Constraints. In *Proceedings of the 9th International Conference on Database Theory*, 2003. to appear.
3. N. Alon, T. Milo, F. Neven, D. Suciu, and V. Vianu. XML with data values: Typechecking revisited. In *Proceedings of the 20th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 138–149, 2001.
4. S. Alstrup and T. Rauhe. Improved labeling scheme for ancestor queries. In *Proceedings of ACM-SIAM Symposium on Discrete Algorithms*, 2002.
5. Sihem Amer-Yahia and Mary Fernandez. Overview of existing XML storage techniques. Technical report, AT&T Labs – Research, 2002. Available as preprint in `http://www.research.att.com/~sihem/SIGRECORD02.pdf`.
6. M. Arenas and L. Libkin. A normal form for XML documents. In *Proceedings of the 21st ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 85–96, 2002.
7. Michael Benedikt, Wenfei Fan, and Gabriel M. Kuper. Structural properties of XPath fragments. International Conference on Database Theory, 2003.
8. Michael Benedikt and Leonid Libkin. Tree Extension Algebras: logics, automata, and query languages. In *17th Annual IEEE Symposium on Logic in Computer Science*, 2002.
9. G. J. Bex, S. Maneth, and F. Neven. A formal model for an expressive fragment of XSLT. *Information Systems*, 27(1):21–39, 2002.
10. Paul V. Biron and Ashok Malhotra. XML Schema Part 2: Datatypes, W3C Recommendation 02 May 2001. Technical report, W3C, 2001. Available at `http://www.w3.org/TR/xmlschema-2/`.
11. S. Boag, D. Chamberlin, M. Fernandez, D. Florescu, J. Robie, and J. Simeon. XQuery 1.0: an XML query language, 2002. Available from the W3C, `http://www.w3.org/TR/xquery`.
12. Bert Bos, Håkon Wium Lie, Chris Lilley, and Ian Jacobs. Cascading Style Sheets, level 2, CSS2 specification. W3C recommendation, World Wide Web Consortium, 1998.
13. C. Brabrand, A. Møller, and M. Schwartzbach. Static validation of dynamically generated HTML. In *Proceedings of the ACM SIGPLAN SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE'01)*, 2001.
14. A. Brüggemann-Klein, S. Hermann, and D. Wood. Context, caterpillars, tree automata, and tree pattern matching. In *Proceedings of the 4th International Conference on Developments in Language Theory*, pages 1–9, 1999.
15. Anne Brüggemann-Klein, Makoto Murata, and Derick Wood. Regular tree and regular hedge languages over unranked alphabets. Technical Report HKUST-TCSC-2001-06, HKUST, 2001.
16. Peter Buneman, Susan B. Davidson, Wenfei Fan, Carmem S. Hara, and Wang Chiew Tan. Keys for XML. In *World Wide Web*, pages 201–210, 2001.
17. D. Calvanese, G. De Giacomo, M. Lenzerini, and M. Y. Vardi. Rewriting of regular expressions and regular path queries. In *Proceedings of the 18th ACM Symposium on Principles of Database Systems*, pages 194–204. ACM Press, 1999.

18. D. Calvanese, G. De Giacomo, M. Lenzerini, and M. Y. Vardi. Answering regular path queries using views. In *Proceedings of the 16th International Conference on Data Engineering*, pages 389–398, 2000.
19. D. Calvanese, G. De Giacomo, M. Lenzerini, and M. Y. Vardi. Containment of conjunctive regular path queries with inverse. In *KR-2000*, pages 176–185, 2000.
20. Diego Calvanese, Giuseppe De Giacomo, Maurizio Lenzerini, and Moshe Y. Vardi. View-based query answering and query containment over semistructured data. In *Proceedings of the 8th Int. Workshop on Database Programming Languages (DBPL 2001)*, 2001.
21. Diego Calvanese, Giuseppe De Giacomo, Maurizio Lenzerini, and Moshe Y. Vardi. View-based query processing for regular path queries with inverse. In *Proceedings of the Nineteenth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, May 15-17, 2000, Dallas, Texas, USA*, pages 58–66. ACM, 2000.
22. Michael J. Carey, Jerry Kiernan, Jayavel Shanmugasundaram, Eugene J. Shekita, and Subbu N. Subramanian. XPERANTO: Middleware for publishing object-relational data as XML documents. In *VLDB 2000, Proceedings of 26th International Conference on Very Large Data Bases*, pages 646–648, 2000.
23. James Clark. XML path language (XPath), 1999. Available from the W3C, `http://www.w3.org/TR/xpath`.
24. James Clark. XSL transformations (XSLT) specification, 1999. Available from the W3C, `http://www.w3.org/TR/WD-xslt`.
25. James Clark and Murata Makoto. Relax NG specification. Technical report, OASIS, 2001.
26. E. Cohen, H. Kaplan, and T. Milo. Labeling dynamic XML trees. In *Proceedings of the 21st ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, 2002.
27. Hubert Comon, Max Dauchet, Remi Gilleron, Florent Jacquemard, Denis Lugiez, Sophie Tison, and Marc Tommasi. *Tree Automata Techniques and Applications*. Internet publication, 2002. Available at `http://www.grappa.univ-lille3.fr/tata/`.
28. World Wide Web Consortium. Extensible markup language (XML) 1.0, 1998. `http://www.w3.org/TR/REC-xml`.
29. John Cowan and Richard Tobin. XML Information Set, W3C Recommendation 24 October 2001. Technical report, W3C, 2001. Available as `http://www.w3.org/TR/xml-infoset/`.
30. A. Deutsch, M. Fernandez, and D. Suciu. Storing semistructured data with STORED. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 431–442, 1999.
31. Alin Deutsch and Val Tannen. Containment and Integrity Constraints for XPath Fragments. In *Proceedings of the 8th International Workshop on Knowledge Representation meets Databases*, 2001.
32. J. Engelfriet and H. J. Hoogeboom. Tree-walking pebble automata. In J. Karhumäki, H. Maurer, G. Paun, and G.Rozenberg, editors, *Jewels are forever, contributions to Theoretical Computer Science in honor of Arto Salomaa*, pages 72–83. Springer-Verlag, 1999.
33. J. Engelfriet, H.J. Hoogenboom, and J.P. Van Best. Trips on trees. *Acta Cybernetica*, 14:51–64, 1999.

34. David C. Fallside. XML Schema Part 0: Primer, W3C Recommendation, 2 May 2001. Technical report, W3C, 2002. Available at `http://www.w3.org/TR/xmlschema-0/`.
35. Wenfei Fan and Leonid Libkin. On XML integrity constraints in the presence of DTDs. *Journal of the ACM*, 49(3):368–406, May 2002.
36. M. Fernandez, Y. Kadiyska, A. Morishima, D. Suciu, and W. Tan. SilkRoute : a framework for publishing relational data in XML. *ACM Transactions on Database Technology*, 27(4), December 2002.
37. M. Fernandez, A. Morishima, and D. Suciu. Efficient evaluation of XML middleware queries. In *Proceedings of ACM SIGMOD Conference on Management of Data*, Santa Barbara, 2001.
38. M. Fernandez, D. Suciu, and W. Tan. SilkRoute: trading between relations and XML. In *Proceedings of the WWW9*, pages 723–746, Amsterdam, 2000.
39. Mary Fernández, Jonathan Marsh, and Marton Nagy. XQuery 1.0 and XPath 2.0 data model, W3C Working Draft 16 august 2002. Technical report, W3C, 2002. Available as `http://www.w3.org/TR/query-datamodel/`.
40. Daniela Florescu and Donald Kossmann. Storing and querying XML data using an RDBMS. *IEEE Data Engineering Bulletin*, 22(3), 1999.
41. Daniela Florescu, Alon Y. Levy, and Dan Suciu. Query containment for conjunctive queries with regular expressions. In *Proceedings of the 17th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, June 1-3, 1998, Seattle, Washington*, pages 139–148. ACM Press, 1998.
42. J. Flum, M. Frick, and M. Grohe. Query evaluation via tree-decompositions. In *Proceedings of the 8th International Conference on Database Theory*, pages 22–38, 2001.
43. M. Frick and M. Grohe. The complexity of first-order and monadic second-order logic revisited. In *Proceedings of the 17th IEEE Symposium on Logic in Computer Science*, 2002.
44. C. F. Goldfarb. *The SGML Handbook*. Clarendon Press, Oxford, 1990.
45. G. Gottlob and C. Koch. Monadic datalog and the expressive power of languages for web information extraction. In *Proceedings of the 21st ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, Madison, Wisconsin*, June 2002.
46. Georg Gottlob and Christoph Koch. Monadic queries over tree-structured data. In *17th Annual IEEE Symposium on Logic in Computer Science*, 2002.
47. Arnaud Le Hors et al. Document Object Model (DOM) level 3 Core Specification, W3C Working Draft 09 april 2002. Technical report, W3C, 2002. Available as `http://www.w3.org/TR/2002/WD-DOM-Level-3-Core-20020409/`.
48. Haruo Hosoya and Benjamin C. Pierce. XDuce: An XML processing language (preliminary report). In *WebDB'2000*, pages 226–244, 2000. http://www.research.att.com/conf/webdb2000/.
49. Jeff Jelliffe. The Schematron: An XML structure validation language using patterns in trees, 1999. Available at `http://www.ascc.net/xml/resource/schematron/schematron.html`.
50. N. D. Jones and S. S. Muchnick. Flow analysis and optimization of LISP-like structures. In *6th Annual ACM Symposium on Principles of Programming Languages*, 1979.
51. N. Klarlund, N. Damgaard, and M.I. Schwartzbach. Yakyak: Parsing with logical side constraints. In G. Rozenberg and W. Thomas, editors, *Develop-*

ments in language theory. Foundations, applications, and perspectives. Aachen, Germany, 6-9 July 1999., pages 286–304. World Scientific, 2000.
52. N. Klarlund and M. Schwartzbach. Graph types. In *Proceedings of the 20th Annual ACM Symposium on Principles of Programming Languages*, pages 196–205. ACM, 1993.
53. Nils Klarlund, Anders Møller, and Michael I. Schwartzbach. The DSD schema language. *Automated Software Engineering*, 9(3):285–319, August 2002. Also available at `http://ipsapp009.lwwonline.com/content/search/4531/22/5/fulltext.pdf`.
54. Dongwon Lee and Wesley W. Chu. Comparative analysis of six XML schema languages. *SIGMOD Record*, 29(3):76–87, 2000.
55. Yanhong A. Liu, Ning Li, and Scott D. Stoller. Solving regular tree grammar based constraints. In *Static Analysis Symposium*, volume 2126 of *Lecture Notes in Computer Science*, 2001.
56. Lixto Project, 2002. Available at `http://www.dbai.tuwien.ac.at/proj/lixto/`.
57. Murata Makoto. Extended path expressions for XML. In *Proceedings of the Twenteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, May 21-23, 2001, Santa Barbara, California, USA*. ACM, 2001.
58. S. Maneth and F. Neven. Structured document transformations based on XSL. In *In Proceedings of the 8th International Workshop on Database Programming Languages*, Scotland, 1999.
59. R. McNaughton and S. Papert. *Counter-Free Automata*. MIT Press, 1971.
60. Gerome Miklau and Dan Suciu. Containment and equivalence for an XPath fragment. In *Proceedings of the 21st ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, 2002.
61. T. Milo, D. Suciu, and V. Vianu. Typechecking for XML transformers. In *Proceedings of the ACM Symposium on Principles of Database Systems*, pages 11–22, Dallas, TX, 2000.
62. Makoto Murata. Transformation of documents and schemas by patterns and contextual conditions. In C. Nicholas and D. Wood, editors, *Proceedings of Third International Workshop on Principles of Document Processing (PODP)*, pages 153–169, Palo Alto, CA, September 1996.
63. A. Neumann and H. Seidl. Locating matches of tree patterns in forests. In V. Arvind and R. Ramanujam, editors, *Foundations of Software Technology and Theoretical Computer Science*, Lecture Notes in Computer Science, pages 134–145. Springer, 1998.
64. F. Neven and T. Schwentick. Query automata. In *Proceedings of the 18th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 205–214, 1999.
65. F. Neven and T. Schwentick. Expressive and efficient pattern languages for tree-structured data. In *Proceedings of the 19th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 145–156, 2000.
66. F. Neven and T. Schwentick. On the power of tree-walking automata. In *Proceedings of the 27th International Colloquium on Automata, Languages, and Programming (ICALP 2000), Genf*, pages 547–560, 2000.
67. F. Neven and T. Schwentick. XPath containment in the presence of disjunction, DTDs, and variables. In *Proceedings of the 9th International Conference on Database Theory*, 2003. to appear.

68. F. Neven and J. Van den Bussche. Expressiveness of structured document query languages based on attribute grammars. *Journal of the ACM*, 49(1):56–100, 2002.
69. Frank Neven. Automata, logic, and XML. In *Computer Science Logic*, pages 2–26, 2002.
70. M. O. Rabin. Decidability of second-order theories and automata on infinite trees. *Transactions of the AMS*, 141:1–35, 1969.
71. Dave Raggett. Assertion grammars. Available at `http://www.w3.org/People/Raggett/dtdgen/Docs/`, 1999.
72. Luc Segoufin and Victor Vianu. Validating streaming XML documents. In *Proceedings of the 21st ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, 2002.
73. J. Shanmugasundaram, K. Tufte, G. He, C. Zhang, D. DeWitt, and J. Naughton. Relational databases for querying XML documents: limitations and opportunities. In *Proceedings of 25th International Conference on Very Large Data Bases*, pages 302–314, Edinburgh, UK, September 1999.
74. Jayavel Shanmugasundaram, Eugene J. Shekita, Jerry Kiernan, Rajasekar Krishnamurthy, Stratis Viglas, Jeffrey F. Naughton, and Igor Tatarinov. A general techniques for querying XML documents using a relational database system. *SIGMOD Record*, 30(3):20–26, 2001.
75. Jerome Simeon and Philip Wadler. The essence of XML. In *Sixth International Symposium on Functional and Logic Programming*, 2002.
76. A. Snoeren, K. Conley, and D. Gifford. Mesh-based content routing using XML. In *Proceedings of the 18th Symposium on Operating Systems Principles*, 2001.
77. D. Suciu. Typechecking for semistructured data. In *Proceedings of the International Workshop on Database Programming Languages*, Italy, September 2001. Springer Verlag. (to appear).
78. I. Tatarinov, S.D. Viglas, K. Beyer, J. Shanmugasundaram, E.Shekita, and C. Zhang. Storing and querying ordered XML using a relational database system. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data*. ACM, 2002.
79. H. Thompson, D. Beech, M. Maloney, and N. Mendelsohn. XML Schema part 1: Structures, May 2001. `http://www.w3.org/TR/xmlschema-1/`.
80. Henry S. Thompson, David Beech, Murray Maloney, and Noah Mendelsohn. XML Schema Part 1: Structures W3C Recommendation 2 May 2001. Technical report, W3C, 2002. Available at `http://www.w3.org/TR/xmlschema-1/`.
81. Akihiko Tozawa. Towards static type inference for XSLT. In *Proceedings of the ACM Symposium on Document Engineering*, 2001.
82. Philip Wadler. A formal semantics of patterns in XSLT. In *Markup Languages*. MIT Press, 2001.
83. M. Yoshikawa, T. Amagasa, T. Shimura, and S. Uemura. XRel: a path-based approach to storage and retrieval of XML documents using relational databases. *ACM Transactions on Internet Technology*, 1(1):110–141, 2001.