A Semantic Web Primer

second edition

Grigoris Antoniou

and

Frank van Harmelen

The MIT Press
Cambridge, Massachusetts
London, England

2 *Structured Web Documents: XML*

2.1 Introduction

Today HTML (hypertext markup language) is the standard language in which Web pages are written. HTML, in turn, was derived from SGML (standard generalized markup language), an international standard (ISO 8879) for the definition of device- and system-independent methods of representing information, both human- and machine-readable. Such standards are important because they enable effective communication, thus supporting technological progress and business collaboration. In the WWW area, standards are set by the W3C (World Wide Web Consortium); they are called *recommendations*, in acknowledgment of the fact that in a distributed environment without central authority, standards cannot be enforced.

Languages conforming to SGML are called SGML applications. HTML is such an application; it was developed because SGML was considered far too complex for Internet-related purposes. XML (extensible markup language) is another SGML application, and its development was driven by shortcomings of HTML. We can work out some of the motivations for XML by considering a simple example, a Web page that contains information about a particular book.

```
<h2>Nonmonotonic Reasoning:  Context-Dependent
   Reasoning</h2>
<i>by <b>V. Marek</b> and <b>M. Truszczynski</b></i><br>
Springer 1993<br>
ISBN 0387976892
```

A typical XML representation of the same information might look like this:

```
<book>
  <title>
    Nonmonotonic Reasoning: Context-Dependent Reasoning
  </title>
  <author>V. Marek</author>
  <author>M. Truszczynski</author>
  <publisher>Springer</publisher>
  <year>1993</year>
  <ISBN>0387976892</ISBN>
</bock>
```

Before we turn to differences between the HTML and XML representations, let us observe a few similarities. First, both representations use *tags*, such as `<h2>` and `</year>`. Indeed both HTML and XML are *markup languages*: they allow one to write some content and provide information about what role that content plays.

Like HTML, XML is based on tags. These tags may be nested (tags within tags). All tags in XML must be closed (for example, for an opening tag `<title>` there must be a closing tag `</title>`), whereas in HTML some tags, such as `<br>`, may be left open. The enclosed content, together with its opening and closing tags, is referred to as an *element*. (The recent development of XHTML has brought HTML more in line with XML: any valid XHTML document is also a valid XML document, and as a consequence, opening and closing tags in XHTML are balanced.)

A less formal observation is that human users can read both HTML and XML representations quite easily. Both languages were designed to be easily understandable and usable by humans. But how about machines? Imagine an intelligent agent trying to retrieve the names of the authors of the book in the previous example. Suppose the HTML page could be located with a Web search (something that is not at all clear; the limitations of current search engines are well documented). There is no *explicit* information as to who the authors are. A reasonable guess would be that the authors' names appear immediately after the title or immediately follow the word *by*. But there is no guarantee that these conventions are always followed. And even if they were, are there two authors, "V. Marek" and "M. Truszczynski", or just one, called "V. Marek and M. Truszczynski"? Clearly, more text processing is needed to answer this question, processing that is open to errors.

The problems arise from the fact that the HTML document does not contain structural information, that is, information about pieces of the document and their relationships. In contrast, the XML document is far more eas-

ily accessible to machines because every piece of information is described. Moreover, some forms of their *relations* are also defined through the nesting structure. For example, the <author> tags appear within the <book> tags, so they describe properties of the particular book. A machine processing the XML document would be able to deduce that the author element refers to the enclosing book element, rather than having to infer this fact from proximity considerations, as in HTML. An additional advantage is that XML allows the definition of constraints on values (for example, that a year must be a number of four digits, that the number must be less than 3,000). *XML allows the representation of information that is also machine-accessible.*

Of course, we must admit that the HTML representation provides more than the XML representation: the formatting of the document is also described. However, this feature is not a strength but a weakness of HTML: it *must* specify the formatting; in fact, the main use of an HTML document is to display information (apart from linking to other documents). On the other hand, *XML separates content from formatting.* The same information can be displayed in different ways without requiring multiple copies of the same content; moreover, the content may be used for purposes other than display.

Let us now consider another example, a famous law of physics. Consider the HTML text

```
<h2>Relationship force-mass</h2>
<i>F = M × a</i>
```

and the XML representation

```
<equation>
  <meaning>Relationship force-mass</meaning>
  <leftside>F</leftside>
  <rightside>M × a</rightside>
</equation>
```

If we compare the HTML document to the previous HTML document, we notice that both use basically the same tags. That is not surprising, since they are *predefined.* In contrast, the second XML document uses completely different tags from the first XML document. This observation is related to the intended use of representations. HTML representations are intended to display information, so the set of tags is fixed: lists, bold, color, and so on. In XML we may use information in various ways, and it is up to the user to define a vocabulary suitable for the application. Therefore, *XML is a metalanguage for markup: it does not have a fixed set of tags but allows users to define tags of their own.*

Just as people cannot communicate effectively if they don't use a common language, applications on the WWW must agree on common vocabularies if they need to communicate and collaborate. Communities and business sectors are in the process of defining their specialized vocabularies, creating XML applications (or extensions; thus the term *extensible* in the name of XML). Such XML applications have been defined in various domains, for example, mathematics (MathML), bioinformatics (BSML), human resources (HRML), astronomy (AML), news (NewsML), and investment (IRML).

Also, the W3C has defined various languages on top of XML, such as SVG and SMIL. This approach has also been taken for RDF (see chapter 3).

It should be noted that XML can serve as a *uniform data exchange format* between applications. In fact, XML's use as a data exchange format between applications nowadays far outstrips its originally intended use as document markup language. Companies often need to retrieve information from their customers and business partners, and update their corporate databases accordingly. If there is not an agreed common standard like XML, then specialized processing and querying software must be developed for each partner separately, leading to technical overhead; moreover, the software must be updated every time a partner decides to change its own database format.

Chapter Overview

This chapter presents the main features of XML and associated languages. It is organized as follows:

- Section 2.2 describes the XML language in more detail, and section 2.3 describes the structuring of XML documents.
- In relational databases the structure of tables must be defined. Similarly, the structure of an XML document must be defined. This can be done by writing a DTD (Document Type Definition), the older approach, or an XML schema, the modern approach that will gradually replace DTDs.
- Section 2.4 describes namespaces, which support the modularization of DTDs and XML schemas.
- Section 2.5 is devoted to the accessing and querying of XML documents, using XPath.
- Finally, section 2.6 shows how XML documents can be transformed to be displayed (or for other purposes) using XSL and XSLT.

2.2 The XML Language

An *XML document* consists of a prolog, a number of elements, and an optional epilog (not discussed here).

2.2.1 Prolog

The prolog consists of an XML declaration and an optional reference to external structuring documents. Here is an example of an *XML declaration*:

```
<?xml version="1.0" encoding="UTF-16"?>
```

It specifies that the current document is an XML document, and defines the version and the character encoding used in the particular system (such as UTF-8, UTF-16, and ISO 8859-1). The character encoding is not mandatory, but its specification is considered good practice. Sometimes we also specify whether the document is self-contained, that is, whether it does not refer to external structuring documents:

```
<?xml version="1.0" encoding="UTF-16" standalone="no"?>
```

A reference to external structuring documents looks like this:

```
<!DOCTYPE book SYSTEM "book.dtd">
```

Here the structuring information is found in a local file called `book.dtd`. Instead, the reference might be a URL. If only a locally recognized name or only a URL is used, then the label `SYSTEM` is used. If, however, one wishes to give both a local name and a URL, then the label `PUBLIC` should be used instead.

2.2.2 Elements

XML elements represent the "things" the XML document talks about, such as books, authors, and publishers. They compose the main concept of XML documents. An element consists of an *opening tag*, its *content*, and a *closing tag*. For example,

```
<lecturer>David Billington</lecturer>
```

Tag names can be chosen almost freely; there are very few restrictions. The most important ones are that the first character must be a letter, an underscore, or a colon; and that no name may begin with the string "xml" in any combination of cases (such as "Xml" and "xML").

The content may be text, or other elements, or nothing. For example,

```
<lecturer>
  <name>David Billington</name>
  <phone>+61-7-3875 507</phone>
</lecturer>
```

If there is no content, then the element is called *empty*. An empty element like

```
<lecturer></lecturer>
```

can be abbreviated as

```
<lecturer/>
```

2.2.3 Attributes

An empty element is not necessarily meaningless, because it may have some properties in terms of *attributes*. An attribute is a name-value pair inside the opening tag of an element:

```
<lecturer name="David Billington" phone="+61-7-3875 507"/>
```

Here is an example of attributes for a nonempty element:

```
<order orderNo="23456" customer="John Smith"
       date="October 15, 2002">
  <item itemNo="a528" quantity="1"/>
  <item itemNo="c817" quantity="3"/>
</order>
```

The same information could have been written as follows, replacing attributes by nested elements:

```
<order>
  <orderNo>23456</orderNo>
  <customer>John Smith</customer>
  <date>October 15, 2002</date>
  <item>
    <itemNo>a528</itemNo>
    <quantity>1</quantity>
  </item>
```

```
    <item>
      <itemNo>c817</itemNo>
      <quantity>3</quantity>
    </item>
  </order>
```

When to use elements and when attributes is often a matter of taste. However, note that attributes cannot be nested.

2.2.4 Comments

A comment is a piece of text that is to be ignored by the parser. It has the form

```
<!-- This is a comment -->
```

2.2.5 Processing Instructions (PIs)

PIs provide a mechanism for passing information to an application about how to handle elements. The general form is

```
<?target instruction?>
```

For example,

```
<?stylesheet type="text/css" href="mystyle.css"?>
```

PIs offer procedural possibilities in an otherwise declarative environment.

2.2.6 Well-Formed XML Documents

An XML document is well-formed if it is syntactically correct. The following are some syntactic rules:

- There is only one outermost element in the document (called the *root element*).
- Each element contains an opening and a corresponding closing tag.
- Tags may not overlap, as in `<author><name>Lee Hong</author></name>`.
- Attributes within an element have unique names.
- Element and tag names must be permissible.

2.2.7 Tree Model of XML Documents

It is possible to represent well-formed XML documents as trees; thus trees provide a formal data model for XML. This representation is often instructive. As an example, consider the following document:

```
<?xml version="1.0" encoding="UTF-16"?>
<!DOCTYPE email SYSTEM "email.dtd">
<email>
  <head>
    <from name="Michael Maher"
          address="michaelmaher@cs.gu.edu.au"/>
    <to   name="Grigoris Antoniou"
          address="grigoris@cs.unibremen.de"/>
    <subject>Where is your draft?</subject>
  </head>
  <body>
  Grigoris, where is the draft of the paper
  you promised me last week?
  </body>
</email>
```

Figure 2.1 shows the tree representation of this XML document. It is an ordered labeled tree:

- There is exactly one root.
- There are no cycles.
- Each node, other than the root, has exactly one parent.
- Each node has a label.
- The order of elements is important.

However, whereas the order of elements is important, the order of attributes is not. So, the following two elements are equivalent:

```
<person lastname="Woo" firstname="Jason"/>
<person firstname="Jason" lastname="Woo"/>
```

This aspect is not represented properly in the tree. In general, we would require a more refined tree concept; for example, we should also differentiate between the different types of nodes (element node, attribute node, etc.).

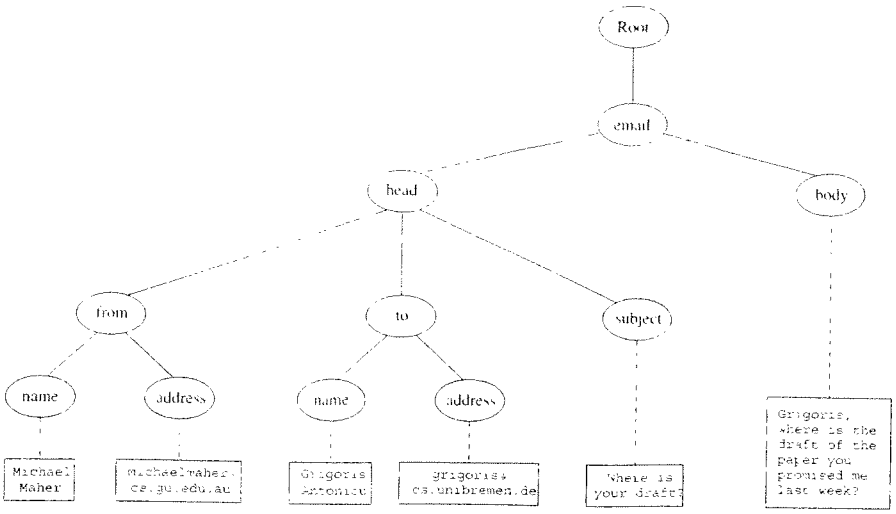

Figure 2.1 Tree representation of an XML document

However, here we use graphs as illustrations, so we do not go into further detail.

Figure 2.1 also shows the difference between the *root* (representing the XML document), and the *root element*, in our case the `email` element. This distinction will play a role when we discuss addressing and querying XML documents in section 2.5.

2.3 Structuring

An XML document is well-formed if it respects certain syntactic rules. However, those rules say nothing specific about the structure of the document. Now, imagine two applications that try to communicate, and that they wish to use the same vocabulary. For this purpose it is necessary to define all the element and attribute names that may be used. Moreover, the structure should also be defined: what values an attribute may take, which elements may or must occur within other elements, and so on.

In the presence of such structuring information we have an enhanced possibility of document validation. We say that an XML document is *valid* if it

is well-formed, uses structuring information, and respects that structuring information.

There are two ways of defining the structure of XML documents: DTDs, the older and more restricted way, and XML Schema, which offers extended possibilities, mainly for the definition of data types.

2.3.1 DTDs

External and Internal DTDs

The components of a DTD can be defined in a separate file (*external DTD*) or within the XML document itself (*internal DTD*). Usually it is better to use external DTDs, because their definitions can be used across several documents; otherwise duplication is inevitable, and the maintenance of consistency over time becomes difficult.

Elements

Consider the element

```
<lecturer>
  <name>David Billington</name>
  <phone>+61-7-3875 507</phone>
</lecturer>
```

from the previous section. A DTD for this element type[1] looks like this:

```
<!ELEMENT lecturer (name,phone)>
<!ELEMENT name (#PCDATA)>
<!ELEMENT phone (#PCDATA)>
```

The meaning of this DTD is as follows:

- The element types `lecturer`, `name`, and `phone` may be used in the document.
- A `lecturer` element contains a `name` element and a `phone` element, in that order.

1. The distinction between the element type `lecturer` and a particular element of this type, such as `David Billington`, should be clear. All particular elements of type `lecturer` (referred to as `lecturer` elements) share the same structure, which is defined here.

- A `name` element and a `phone` element may have any content. In DTDs, `#PCDATA` is the only atomic type for elements.

We express that a `lecturer` element contains either a `name` element or a `phone` element as follows:

```
<!ELEMENT lecturer (name|phone)>
```

It gets more difficult when we wish to specify that a `lecturer` element contains a `name` element and a `phone` element *in any order*. We can only use the trick

```
<!ELEMENT lecturer ((name,phone)|(phone,name))>
```

However, this approach suffers from practical limitations (imagine ten elements in any order).

Attributes

Consider the element

```
<order orderNo="23456" customer="John Smith"
       date="October 15, 2002">
  <item itemNo="a528" quantity="1"/>
  <item itemNo="c817" quantity="3"/>
</order>
```

from the previous section. A DTD for it looks like this:

```
<!ELEMENT order (item+)>
<!ATTLIST order
  orderNo   ID     #REQUIRED
  customer  CDATA  #REQUIRED
  date      CDATA  #REQUIRED>
<!ELEMENT item EMPTY>
<!ATTLIST item
  itemNo    ID     #REQUIRED
  quantity  CDATA  #REQUIRED
  comments  CDATA  #IMPLIED>
```

Compared to the previous example, a new aspect is that the `item` element type is defined to be empty. Another new aspect is the appearance of + after `item` in the definition of the `order` element type. It is one of the *cardinality operators*:

?: appears zero times or once.

*: appears zero or more times.

+: appears one or more times.

No cardinality operator means exactly once.

In addition to defining elements, we have to define attributes. This is done in an *attribute list*. The first component is the name of the element type to which the list applies, followed by a list of triplets of attribute name, attribute type, and value type. An *attribute* name is a name that may be used in an XML document using a DTD.

Attribute Types

Attribute types are similar to predefined data types, but the selection is very limited. The most important attribute types are

- `CDATA`, a string (sequence of characters),
- `ID`, a name that is unique across the entire XML document,
- `IDREF`, a reference to another element with an ID attribute carrying the same value as the IDREF attribute,
- `IDREFS`, a series of IDREFs,
- $(v_1|\ldots|v_n)$, an enumeration of all possible values.

The selection is not satisfactory. For example, dates and numbers cannot be specified; they have to be interpreted as strings (`CDATA`); thus their specific structure cannot be enforced.

Value Types

There are four value types:

- `#REQUIRED`. The attribute must appear in every occurrence of the element type in the XML document. In the previous example, `itemNo` and `quantity` must always appear within an `item` element.
- `#IMPLIED`. The appearance of the attribute is optional. In the example, comments are optional

- `#FIXED "value"`. Every element must have this attribute, which has always the value given after `#FIXED` in the DTD. A value given in an XML document is meaningless because it is overridden by the fixed value.
- `"value"`. This specifies the default value for the attribute. If a specific value appears in the XML document, it overrides the default value. For example, the default encoding of the e-mail system may be "`mime`", but "`binhex`" will be used if specified explicitly by the user.

Referencing

Here is an example for the use of `IDREF` and `IDREFS`. First we give a DTD:

```
<!ELEMENT family (person*)>
<!ELEMENT person (name)>
<!ELEMENT name (#PCDATA)>
<!ATTLIST person
  id       ID     #REQUIRED
  mother   IDREF  #IMPLIED
  father   IDREF  #IMPLIED
  children IDREFS #IMPLIED>
```

An XML element that respects this DTD is the following:

```
<family>

  <person id="bob" mother="mary" father="peter">
    <name>Bob Marley</name>
  </person>

  <person id="bridget" mother="mary">
    <name>Bridget Jones</name>
  </person>

  <person id="mary" children="bob bridget">
    <name>Mary Poppins</name>
  </person>

  <person id="peter" children="bob">
    <name>Peter Marley</name>
  </person>

</family>
```

Readers should study the references between `persons`.

XML Entities

An XML entity can play several roles, such as a placeholder for repeatable characters (a type of shorthand), a section of external data (e.g., XML or other), or as a part of a declaration for elements. A typical use of internal entities, and the one used in subsequent chapters of this book, is similar to constants in a programming language. For example, suppose that a document has several copyright notices that refer to the current year. Then it makes sense to declare an entity

```
<!ENTITY thisyear "2007">
```

Then, at each place the current year needs to be included, we can use the entity reference `&thisyear;` instead. That way, updating the year value to "2008" for the whole document will only mean changing the entity declaration.

A Concluding Example

As a final example we give a DTD for the `email` element from section 2.2.7:

```
<!ELEMENT email (head,body)>
<!ELEMENT head (from,to+,cc*,subject)>
<!ELEMENT from EMPTY>
<!ATTLIST from
  name     CDATA     #IMPLIED
  address  CDATA     #REQUIRED>
<!ELEMENT to EMPTY>
<!ATTLIST to
  name     CDATA     #IMPLIED
  address  CDATA     #REQUIRED>
<!ELEMENT cc EMPTY>
<!ATTLIST cc
  name     CDATA     #IMPLIED
  address  CDATA     #REQUIRED>
<!ELEMENT subject (#PCDATA)>
<!ELEMENT body (text,attachment*)>
<!ELEMENT text (#PCDATA)>
<!ELEMENT attachment EMPTY>
<!ATTLIST attachment
```

```
encoding (mime|binhex) "mime"
file     CDATA     #REQUIRED>
```

We go through some interesting parts of this DTD:

- A `head` element contains a `from` element, at least one `to` element, zero or more `cc` elements, and a `subject` element, in that order.
- In `from`, `to`, and `cc` elements the `name` attribute is not required; the `address` attribute, on the other hand, is always required.
- A `body` element contains a `text` element, possibly followed by a number of `attachment` elements.
- The `encoding` attribute of an `attachment` element must have either the value "`mime`" or "`binhex`", the former being the default value.

We conclude with two more remarks on DTDs. First, a DTD can be interpreted as an Extended Backus-Naur Form (EBNF). For example, the declaration

```
<!ELEMENT email (head,body)>
```

is equivalent to the rule

```
email ::= head body
```

which means that an e-mail consists of a head followed by a body. And second, recursive definitions are possible in DTDs. For example,

```
<!ELEMENT bintree ((bintree root bintree)|emptytree)>
```

defines binary trees: a binary tree is the empty tree, or consists of a left subtree, a root, and a right subtree.

2.3.2 XML Schema

XML Schema offers a significantly richer language for defining the structure of XML documents. One of its characteristics is that its syntax is based on XML itself. This design decision provides a significant improvement in readability, but more important, it also allows significant reuse of technology. It is no longer necessary to write separate parsers, editors, pretty printers, and so on, to obtain a separate syntax, as was required for DTDs; any XML will

do. An even more important improvement is the possibility of reusing and refining schemas. XML Schema allows one to define new types by extending or restricting already existing ones. In combination with an XML-based syntax, this feature allows one to build schemas from other schemas, thus reducing the workload. Finally, XML Schema provides a sophisticated set of data types that can be used in XML documents (DTDs were limited to strings only).

An XML schema is an element with an opening tag like

```
<xsd:schema
  xmlns:xsd="http://www.w3.org/2000/10/XMLSchema"
  version="1.0">
```

The element uses the schema of XML Schema found at the W3C Web site. It is, so to speak, the foundation on which new schemas can be built. The prefix `xsd` denotes the namespace of that schema (more on namespaces in the next section). If the prefix is omitted in the `xmlns` attribute, then we are using elements from this namespace by default:

```
<schema
  xmlns="http://www.w3.org/2000/10/XMLSchema"
  version="1.0">
```

In the following we omit the `xsd` prefix.

Now we turn to `schema` elements. Their most important contents are the definitions of element and attribute types, which are defined using data types.

Element Types

The syntax of element types is

```
<element name="..."/>
```

and they may have a number of optional attributes, such as types,

```
type="..."
```

or cardinality constraints

- `minOccurs="x"`, where `x` may be any natural number (including zero),

- `maxOccurs="x"`, where `x` may be any natural number (including zero) or `unbounded`.

`minOccurs` and `maxOccurs` are generalizations of the cardinality operators `?`, `*`, and `+`, offered by DTDs. When cardinality constraints are not provided explicitly, `minOccurs` and `maxOccurs` have value 1 by default.

Here are a few examples.

```
<element name="email"/>

<element name="head" minOccurs="1" maxOccurs="1"/>

<element name="to" minOccurs="1"/>
```

Attribute Types

The syntax of attribute types is

```
<attribute name="..."/>
```

and they may have a number of optional attributes, such as types,

```
type="..."
```

or existence (corresponds to `#OPTIONAL` and `#IMPLIED` in DTDs),

`use="x"`, where `x` may be `optional` or `required` or `prohibited`,

or a default value (corresponds to `#FIXED` and default values in DTDs).

Here are examples:

```
<attribute name="id" type="ID" use="required"/>

<attribute name="speaks" type="Language" use="optional"
    default="en"/>
```

Data Types

We have already recognized the very restricted selection of data types as a key weakness of DTDs. XML Schema provides powerful capabilities for defining data type. First there is a variety of *built-in data types*. Here we list a few:

- Numerical data types, including `integer`, `short`, `Byte`, `long`, `float`, `decimal`
- String data types, including `string`, `ID`, `IDREF`, `CDATA`, `language`
- Date and time data types, including `time`, `date`, `gMonth`, `gYear`

There are also *user-defined data types*, comprising *simple data types*, which cannot use elements or attributes, and *complex data types*, which can use elements and attributes. We discuss complex types first, deferring discussion of simple data types until we talk about restriction. Complex types are defined from already existing data types by defining some attributes (if any) and using

- `sequence`, a sequence of existing data type elements, the appearance of which in a predefined order is important,
- `all`, a collection of elements that must appear but the order of which is not important,
- `choice`, a collection of elements, of which one will be chosen.

Here is an example:

```
<complexType name="lecturerType">
  <sequence>
    <element name="firstname" type="string"
      minOccurs="0" maxOccurs="unbounded"/>
    <element name="lastname" type="string"/>
  </sequence>
  <attribute name="title" type="string" use="optional"/>
</complexType>
```

The meaning is that an element in an XML document that is declared to be of type `lecturerType` may have a `title` attribute; it may also include any number of `firstname` elements and must include exactly one `lastname` element.

Data Type Extension

Already existing data types can be extended by new elements or attributes. As an example, we extend the `lecturer` data type.

```
<complexType name="extendedLecturerType">
  <extension base="lecturerType">
    <sequence>
      <element name="email" type="string"
        minOccurs="0" maxOccurs="1"/>
    </sequence>
    <attribute name="rank" type="string" use="required"/>
  </extension>
</complexType>
```

In this example, `lecturerType` is extended by an `email` element and a `rank` attribute. The resulting data type looks like this:

```
<complexType name="extendedLecturerType">
  <sequence>
    <element name="firstname" type="string"
      minOccurs="0" maxOccurs="unbounded"/>
    <element name="lastname" type="string"/>
    <element name="email" type="string"
      minOccurs="0" maxOccurs="1"/>
  </sequence>
  <attribute name="title" type="string" use="optional"/>
  <attribute name="rank" type="string" use="required"/>
</complexType>
```

A hierarchical relationship exists between the original and the extended type. *Instances of the extended type are also instances of the original type.* They may contain additional information, but neither less information nor information of the wrong type.

Data Type Restriction

An existing data type may also be restricted by adding constraints on certain values. For example, new `type` and `use` attributes may be added, or the numerical constraints of `minOccurs` and `maxOccurs` tightened.

It is important to understand that restriction is *not* the opposite process from extension. Restriction is not achieved by deleting elements or attributes. Therefore, the following hierarchical relationship still holds: *Instances of the restricted type are also instances of the original type.* They satisfy at least the constraints of the original type and some new ones.

As an example, we restrict the `lecturer` data type as follows:

```
<complexType name="restrictedLecturerType">
  <restriction base="lecturerType">
    <sequence>
      <element name="firstname" type="string"
        minOccurs="1" maxOccurs="2"/>
    </sequence>
    <attribute name="title" type="string" use="required"/>
  </restriction>
</complexType>
```

The tightened constraints are shown in boldface. Readers should compare them with the original ones.

Simple data types can also be defined by restricting existing data types. For example, we can define a type `dayOfMonth` that admits values from 1 to 31 as follows:

```
<simpleType name="dayOfMonth">
  <restriction base="integer">
    <minInclusive value="1"/>
    <maxInclusive value="31"/>
  </restriction>
</simpleType>
```

It is also possible to define a data type by listing all the possible values. For example, we can define a data type `dayOfWeek` as follows:

```
<simpleType name="dayOfWeek">
  <restriction base="string">
    <enumeration value="Mon"/>
    <enumeration value="Tue"/>
    <enumeration value="Wed"/>
    <enumeration value="Thu"/>
    <enumeration value="Fri"/>
    <enumeration value="Sat"/>
    <enumeration value="Sun"/>
  </restriction>
</simpleType>
```

A Concluding Example

Here we define an XML schema for `email`, so that it can be compared to the DTD provided on page 38.

```
<element name="email" type="emailType"/>
<complexType name="emailType">
  <sequence>
    <element name="head" type="headType"/>
    <element name="body" type="bodyType"/>
  </sequence>
</complexType>

<complexType name="headType">
  <sequence>
    <element name="from" type="nameAddress"/>
    <element name="to" type="nameAddress"
      minOccurs="1" maxOccurs="unbounded"/>
    <element name="cc" type="nameAddress"
      minOccurs="0" maxOccurs="unbounded"/>
    <element name="subject" type="string"/>
  </sequence>
</complexType>

<complexType name="nameAddress">
  <attribute name="name" type="string" use="optional"/>
  <attribute name="address" type="string" use="required"/>
</complexType>

<complexType name="bodyType">
  <sequence>
    <element name="text" type="string"/>
    <element name="attachment" minOccurs="0"
        maxOccurs="unbounded">
      <complexType>
        <attribute name="encoding" use="optional"
            default="mime">
          <simpleType>
            <restriction base="string">
              <enumeration value="mime"/>
              <enumeration value="binhex"/>
            </restriction>
          </simpleType>
        </attribute>
        <attribute name="file" type="string"
            use="required"/>
      </complexType>
```

```
    </element>
  </sequence>
</complexType>
```

Note that some data types are defined separately and given names, while others are defined within other types and defined anonymously (the types for the `attachment` element and the `encoding` attribute). In general, if a type is only used once, it makes sense to define it anonymously for local use. However, this approach reaches its limitations quickly if nesting becomes too deep.

2.4 Namespaces

One of the main advantages of using XML as a universal (meta) markup language is that information from various sources may be accessed; in technical terms, an XML document may use more than one DTD or schema. But since each structuring document was developed independently, *name clashes* appear inevitable. If DTD A and DTD B define an element type e in different ways, a parser that tries to validate an XML document in which an e element appears must be told which DTD to use for validation purposes.

The technical solution is simple: disambiguation is achieved by using a different prefix for each DTD or schema. The prefix is separated from the local name by a colon:

```
prefix:name
```

As an example, consider an (imaginary) joint venture of an Australian university, say, Griffith University, and an American university, say, University of Kentucky, to present a unified view for online students. Each university uses its own terminology, and there are differences. For example, lecturers in the United States are not considered regular faculty, whereas in Australia they are (in fact, they correspond to assistant professors in the United States). The following example shows how disambiguation can be achieved.

```
<?xml version="1.0" encoding="UTF-16"?>
<vu:instructors
    xmlns:vu="http://www.vu.com/empDTD"
    xmlns:gu="http://www.gu.au/empDTD"
    xmlns:uky="http://www.uky.edu/empDTD">
  <uky:faculty
```

```
    uky:name="John Smith"
    uky:department="Computer Science"/>
  <gu:academicStaff
    gu:title="lecturer"
    gu:name="Mate Jones"
    gu:school="Information Technology"/>
</vu:instructors>
```

So, namespaces are declared within an element and can be used in that element and any of its children (elements and attributes). A namespace declaration has the form:

```
xmlns:prefix="location"
```

where location may be the address of the DTD or schema. If a prefix is not specified, as in

```
xmlns="location"
```

then the location is used by default. For example, the previous example is equivalent to the following document:

```
<?xml version="1.0" encoding="UTF-16"?>
<vu:instructors
    xmlns:vu="http://www.vu.com/empDTD"
    xmlns="http://www.gu.au/empDTD"
    xmlns:uky="http://www.uky.edu/empDTD">
  <uky:faculty
    uky:title="assistant professor"
    uky:name="John Smith"
    uky:department="Computer Science"/>
  <academicStaff
    title="lecturer"
    name="Mate Jones"
    school="Information Technology"/>
</vu:instructors>
```

2.5 Addressing and Querying XML Documents

In relational databases, parts of a database can be selected and retrieved using query languages such as SQL. The same is true for XML documents, for which there exist a number of proposals for query languages, such as XQL, XML-QL, and XQuery.

The central concept of XML query languages is a *path expression* that specifies how a node, or a set of nodes, in the tree representation of the XML document can be reached. We introduce path expressions in the form of XPath because they can be used for purposes other than querying, namely, for transforming XML documents.

XPath is a language for addressing parts of an XML document. It operates on the tree data model of XML and has a non-XML syntax. The key concepts are path expressions. They can be

- absolute (starting at the root of the tree); syntactically they begin with the symbol `/`, which refers to the root of the document, situated one level above the root element of the document;
- relative to a context node.

Consider the following XML document:

```
<?xml version="1.0" encoding="UTF-16"?>
<!DOCTYPE library PUBLIC "library.dtd">
<library location="Bremen">
  <author name="Henry Wise">
    <book title="Artificial Intelligence"/>
    <book title="Modern Web Services"/>
    <book title="Theory of Computation"/>
  </author>
  <author name="William Smart">
    <book title="Artificial Intelligence"/>
  </author>
  <author name="Cynthia Singleton">
    <book title="The Semantic Web"/>
    <book title="Browser Technology Revised"/>
  </author>
</library>
```

Its tree representation is shown in figure 2.2.

In the following we illustrate the capabilities of XPath with a few examples of path expressions.

1. Address all `author` elements.

```
/library/author
```

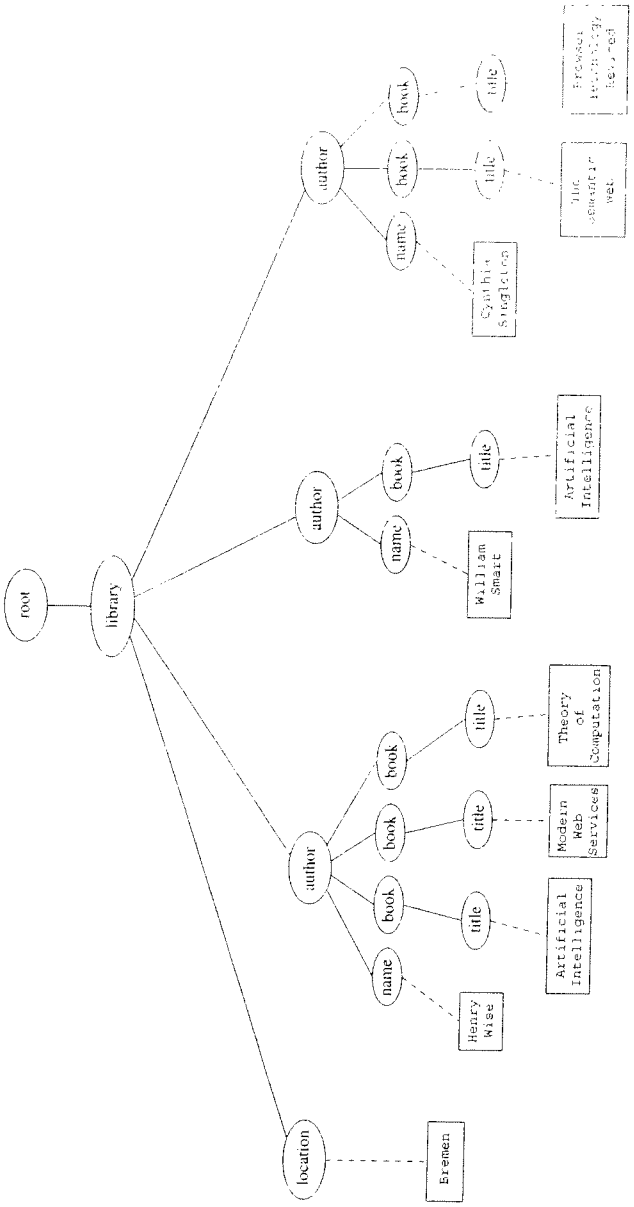

Figure 2.2 Tree representation of a library document

This path expression addresses all `author` elements that are children of the `library` element node, which resides immediately below the root. Using a sequence $/t_1/\ldots/t_n$, where each t_{i+1} is a child node of t_i, we define a path through the tree representation.

2. An alternative solution for the previous example is

```
//author
```

Here `//` says that we should consider all elements in the document and check whether they are of type `author`. In other words, this path expression addresses all `author` elements anywhere in the document. Because of the specific structure of our XML document, this expression and the previous one lead to the same result; however, they may lead to different results, in general.

3. Address the `location` attribute nodes within `library` element nodes.

```
/library/@location
```

The symbol `@` is used to denote attribute nodes.

4. Address all `title` attribute nodes within `book` elements anywhere in the document, which have the value `"Artificial Intelligence"` (see figure 2.3).

```
//book/@title=[.="Artificial Intelligence"]
```

5. Address all books with title `"Artificial Intelligence"` (see figure 2.4).

```
//book[@title="Artificial Intelligence"]
```

We call a test within square brackets a *filter expression*. It restricts the set of addressed nodes.

Note the difference between this expression and the one in query 4. Here we address `book` elements the title of which satisfies a certain condition. In query 4 we collected `title` attribute nodes of `book` elements. A com-

[illegible] figure 2.3 and 2.4 illustrates the difference

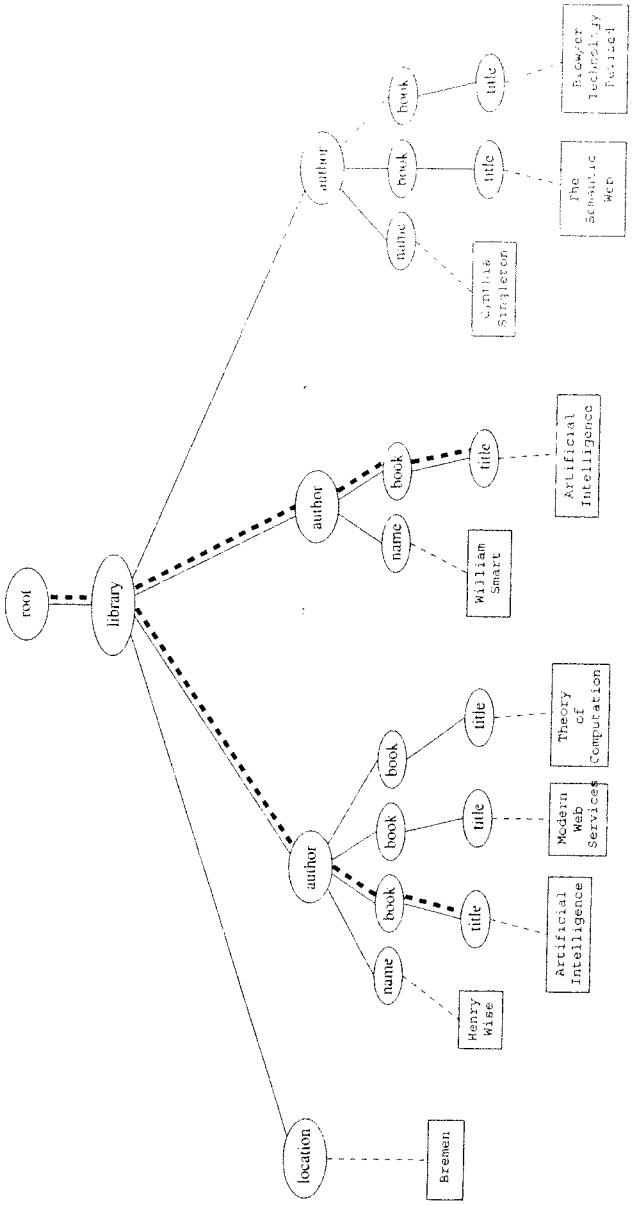

Figure 2.3 Tree representation of query 4

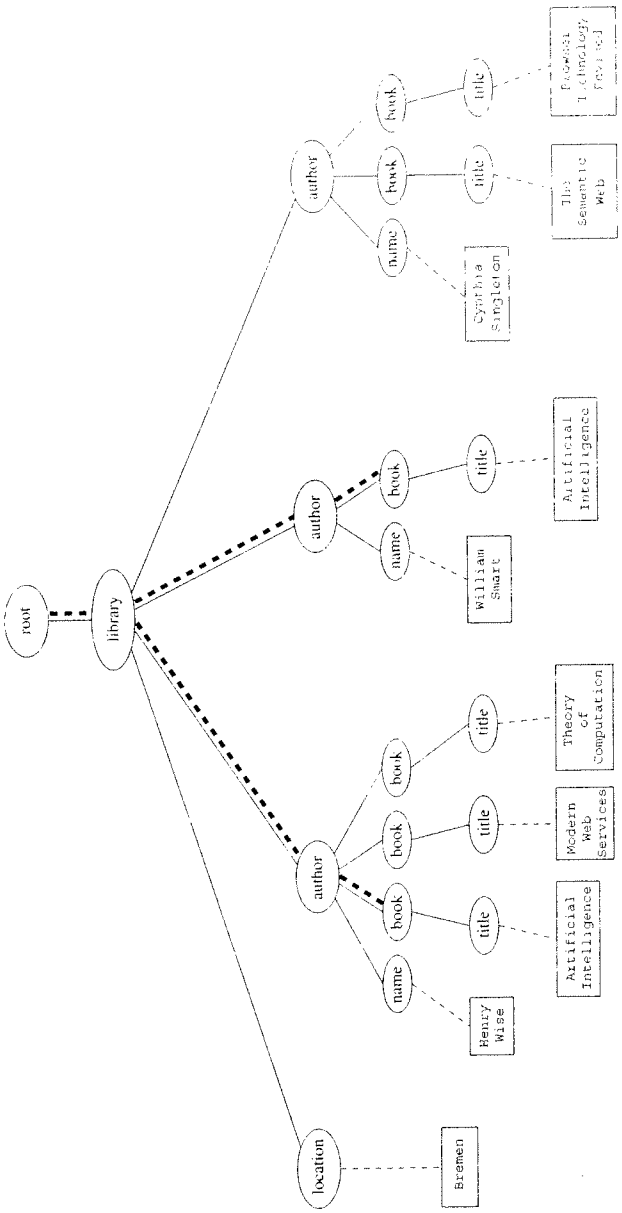

Figure 2.4 Tree representation of query 5

6. Address the first `author` element node in the XML document.

```
//author[1]
```

7. Address the last `book` element within the first `author` element node in the document.

```
//author[1]/book[last()]
```

8. Address all `book` element nodes without a `title` attribute.

```
//book[not (@title)]
```

These examples are meant to give a feeling of the expressive power of path expressions. In general, a path expression consists of a series of steps, separated by slashes. A *step* consists of an axis specifier, a node test, and an optional predicate.

- An *axis specifier* determines the tree relationship between the nodes to be addressed and the context node. Examples are parent, ancestor, child (the default), sibling, attribute node. `//` is such an axis specifier; it denotes descendant or self.
- A *node test* specifies which nodes to address. The most common node tests are element names (which may use namespace information), but there are others. For example, `*` addresses all element nodes, `comment()` all comment nodes, and so on.
- *Predicates* (or *filter expressions*) are optional and are used to refine the set of addressed nodes. For example, the expression `[1]` selects the first node, `[position()=last()]` selects the last node, `[position() mod 2 = 0]` the even nodes, and so on.

We have only presented the abbreviated syntax, XPath actually has a more complicated full syntax. References are found at the end of this chapter.

2.6 Processing

So far we have not provided any information about how XML documents can be displayed. Such information is necessary because unlike HTML documents, XML documents do not contain formatting information. The advantage is that a given XML document can be presented in various ways, when different *style sheets* are applied to it. For example, consider the XML element

```
<author>
  <name>Grigoris Antoniou</name>
  <affiliation>University of Bremen</affiliation>
  <email>ga@tzi.de</email>
</author>
```

The output might look like the following, if a style sheet is used:

Grigoris Antoniou

University of Bremen

ga@tzi.de

Or it might appear as follows, if a different style sheet is used:

Grigoris Antoniou

University of Bremen

ga@tzi.de

Style sheets can be written in various languages, for example, in CSS2 (cascading style sheets level 2). The other possibility is XSL (extensible stylesheet language).

XSL includes both a transformation language (XSLT) and a formatting language. Each of these is, of course, an XML application. XSLT specifies rules with which an input XML document is transformed to another XML document, an HTML document, or plain text. The output document may use the same DTD or schema as the input document, or it may use a completely different vocabulary.

XSLT (XSL transformations) can be used independently of the formatting language. Its ability to move data and metadata from one XML representation to another makes it a most valuable tool for XML-based applications. Generally XSLT is chosen when applications that use different DTDs or schemas need to communicate. XSLT is a tool that can be used for machine-processing of content without any regard to displaying the information for people to read. Despite this fact, in the following we use XSLT only to display XML documents.

One way of defining the presentation of an XML document is to transform it into an HTML document. Here is an example. We define an XSLT document that will be applied to the author example.

```
<?xml version="1.0" encoding="UTF-16"?>
<xsl:stylesheet version="1.0"
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:template match="/author">
    <html>
      <head><title>An author</title></head>
      <body bgcolor="white">
        <b><xsl:value-of select="name"/></b><br></br>
        <xsl:value-of select="affiliation"/><br></br>
        <i><xsl:value-of select="email"/></i>
      </body>
    </html>
  </xsl:template>
</xsl:stylesheet>
```

The output of this style sheet, applied to the previous XML document, produces the following HTML document (which now defines the presentation):

```
<html>
  <head><title>An author</title></head>
  <body bgcolor="white">
    <b>Grigoris Antoniou</b><br>
    University of Bremen<br>
    <i>ga@tzi.de</i>
  </body>
</html>
```

Let us make a few observations. XSLT documents are XML documents. So XSLT resides on top of XML (that is, it is an XML application). The XSLT document defines a *template*; in this case an HTML document, with some placeholders for content to be inserted (see figure 2.5).

In the previous XSLT document, `xsl:value-of` retrieves the value of an element and copies it into the output document. That is, it places some content into the template.

Now suppose we had an XML document with details of several authors. It would clearly be a waste of effort to treat each `author` element separately. In such cases, a special template is defined for `author` elements, which is used by the main template. We illustrate this approach referring to the following input document:

```
<authors>
```

```
<html>
<head><title>An author</title></head>
<body bgcolor="white">
  <b>...</b><br>
  ...<br>
  <i>...</i>
</body>
</html>
```

Figure 2.5 A template

```
  <author>
    <name>Grigoris Antoniou</name>
    <affiliation>University of Bremen</affiliation>
    <email>ga@tzi.de</email>
  </author>
  <author>
    <name>David Billington</name>
    <affiliation>Griffith University</affiliation>
    <email>david@gu.edu.net</email>
  </author>
</authors>
```

We define the following XSLT document:

```
<?xml version="1.0" encoding="UTF-16"?>
<xsl:stylesheet version="1.0"
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:template match="/">
    <html>
    <head><title>Authors</title></head>
    <body bgcolor="white">
      <xsl:apply-templates select="authors"/>
      <!-- Apply templates for AUTHORS children -->
    </body>
    </html>
  </xsl:template>
```

```
    <xsl:template match="authors">
      <xsl:apply-templates select="author"/>
    </xsl:template>

    <xsl:template match="author">
      <h2><xsl:value-of select="name"/></h2>
      Affiliation:<xsl:value-of select="affiliation"/><br>
      Email:  <xsl:value-of select="email"/>
      <p>
    </xsl:template>
  </xsl:stylesheet>
```

The output produced is

```
  <html>
  <head><title>Authors</title></head>
  <body bgcolor="white">
    <h2>Grigoris Antoniou</h2>
    Affiliation:  University of Bremen<br>
    Email:  ga@tzi.de
    <p>
    <h2>David Billington</h2>
    Affiliation:  Griffith University<br>
    Email:  david@gu.edu.net
    <p>
  </body>
  </html>
```

The `xsl:apply-templates` element causes all children of the context node to be matched against the selected path expression. For example, if the current template applies to / (that is, if the current context node is the root), then the element `xsl:apply-templates` applies to the root element, in this case, the `authors` element (remember that / is located above the root element). And if the current node is the `authors` element, then the element `xsl:apply-templates select="author"` causes the template for the `author` elements to be applied to all `author` children of the `authors` element.

It is good practice to define a template for each element type in the document. Even if no specific processing is applied to certain elements, in our example `authors`, the `xsl:apply-templates` element should be used. That way, we work our way from the root to the leaves of the tree, and all templates are indeed applied.

Now we turn our attention to attributes. Suppose we wish to process the element

```
<person firstname="John" lastname="Woo"/>
```

with XSLT. Let us attempt the easiest task imaginable, a transformation of the element to itself. One might be tempted to write

```
<xsl:template match="person">
  <person
    firstname="<xsl:value-of select="@firstname">"
    lastname="<xsl:value-of select="@lastname">"/>
</xsl:template>
```

However, this is not a well-formed XML document because tags are not allowed within the values of attributes. But the intention is clear; we wish to add attribute values into the template. In XSLT, data enclosed in curly brackets take the place of the `xsl:value-of` element. The correct way to define a template for this example is as follows:

```
<xsl:template match="person">
  <person
    firstname="{@firstname}"
    lastname="{@lastname}"/>
</xsl:template>
```

Finally, we give a transformation example from one XML document to another, which does not specify the display. Again we use the `authors` document as input and define an XSLT document as follows:

```
<?xml version="1.0" encoding="UTF-16"?>
<xsl:stylesheet version="1.0"
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:template match="/">
    <authors>
      <xsl:apply-templates select="authors"/>
    </authors>
  </xsl:template>


  <xsl:template match="authors">
    <xsl:apply-templates select="author"/>
```

```
    </xsl:template>

    <xsl:template match="author">
      <author>
        <name><xsl:value-of select="name"/></name>
        <contact>
          <institute>
            <xsl:value-of select="affiliation"/>
          </institute>
          <email><xsl:value-of select="email"/></email>
        </contact>
      </author>
    </xsl:template>

</xsl:stylesheet>
```

The tree representation of the output document is shown in figure 2.6 to illustrate the tree transformation character of XSLT.

2.7 Summary

- XML is a metalanguage that allows users to define markup for their documents using tags.
- Nesting of tags introduces structure. The structure of documents can be enforced using schemas or DTDs.
- XML separates content and structure from formatting.
- XML is the de facto standard for the representation of structured information on the Web and supports machine processing of information.
- XML supports the exchange of structured information across different applications through markup, structure, and transformations.
- XML is supported by query languages.

Some points discussed in subsequent chapters include

- The nesting of tags does not have standard meaning.
- The semantics of XML documents is not accessible to machines, only to people.

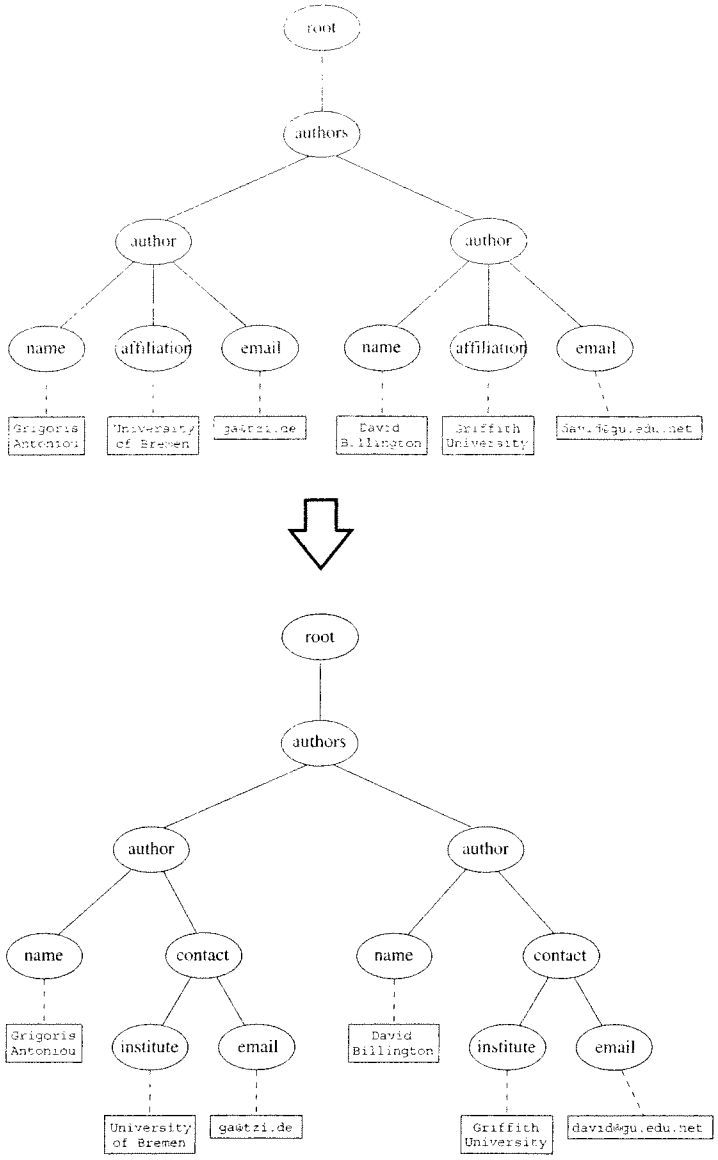

Figure 2.6 XSLT as tree transformation

- Collaboration and exchange are supported if there is an underlying shared understanding of the vocabulary. XML is well-suited for close collaboration, where domain- or community-based vocabularies are used. It is not so well-suited for global communication.

Suggested Reading

Generally the official W3C documents are found at <http://www.w3.org/>. Here we give a few of the most important links, together with some other useful references.

- T. Bray, J. Paoli, C. M. Sperberg-McQueen, E. Maler, and F. Yergeau, eds. Extensible Markup Language (XML) 1.0, 4th ed. W3C Recommendation. September 29, 2006. <http://www.w3.org/TR/REC-xml/>.
- T. Bray, D. Hollander, A. Layman, and R. Tobin, eds. Namespaces in XML 1.0, 2nd ed. W3C Recommendation. August 16, 2006. <http://www.w3.org/TR/REC-xml-names/>.
- J. Clark, and S. DeRose, eds. XML Path Language (XPath) Version 1.0. W3C Recommendation, November 16, 1999. <http://www.w3.org/TR/xpath>.
- A. Berglund, ed. Extensible Stylesheet Language (XSL) Version 1.1. W3C Recommendation. December 5, 2006. <http://www.w3.org/TR/xsl/>.
- J. Clark, ed. XSL Transformations (XSLT) Version 1.0. W3C Recommendation. November 16, 1999. <http://www.w3.org/TR/xslt>.

Recent trends in XML querying may be found at

- <http://www.w3.org/XML/Query/>.

XML has attracted a lot of attention in industry, and many books covering the technicalities in depth exist. Two books for further reading on XML are

- E. R. Harold. *XML 1.1 Bible*, 3rd ed. New York: Wiley 2004.
- W. Willard. *HTML: A Beginner's Guide*, 3rd ed. New York: McGraw Hill (Osborne), 2006.

Several sites have teaching material on XML and related technologies:

- <http://www.xml.com/>, where the following papers may be found:
  - N. Walsh. A Technical Introduction to XML. October 3, 1998.
  - T. Bray. XML Namespaces by Example. January 19, 1999.
  - E. van der Vlist. Using W3C XML Schema. October 17, 2001.
  - G. Holman. What Is XSLT? (I): The Context of XSL Transformations and the XML Path Language. August 16, 2000.
- <http://www.w3schools.com/>.
- <http://www.topxml.com/>.
- <http://www.zvon.org/>.
- <http://www.xslt.com/>.

Exercises and Projects

2.1 In our e-mail example we specified the body of an e-mail to contain exactly one text and a number of attachments. Modify the schema to allow for an arbitrary number of texts and attachments in any order.

2.2 Search the Web for XML applications, with keywords such as "XML DTD" or "XML schema".

2.3 Read the official W3C documents on namespaces, XPath, XSL, and XSLT. Identify some issues that were not covered in this chapter, in particular, the general notation and capabilities of XPath. Write small documents that use these new aspects.

2.4 In this chapter we did not cover links, a crucial ingredient of Web pages. XLink provides linking capabilities that go beyond HTML links. Check out XLink on the official W3C pages. Note that simple links can be created as follows:

```
<mylink xmlns:xlink=http://www.w3.org/1999/xlink"
   xlink:type="simple" xlink:href="target.html">
Click here </mylink>
```

2.5 Discuss the relevance of XSLT for defining *views* on Web sites ("views" hide certain parts of Web sites and display only those parts meant for the particular user's viewing).

2.6 Draw a comparison between document markup using XML and using TeX/LaTeX, also between XML transformations and BibTeX.

For the following projects you are asked to "design a vocabulary". This includes designing a vocabulary, writing a corresponding DTD or schema, writing sample XML documents, and transforming these documents into HTML and viewing them in a Web browser.

2.7 Design a vocabulary to model (parts of) your workplace. For example, if you are at a university, design a vocabulary about courses, teaching staff, rooms, publications, and so on.

2.8 For one of your hobbies, design a vocabulary for exchanging information with others who share your interest.

2.9 Perhaps you read books of certain categories? Design a vocabulary for describing them and communicating about them with other people.

2.10 Are you an investor? Design a vocabulary about available investment options and their properties (for example, risk, return, investor age, investor character).

2.11 Do you like cooking? Design a vocabulary about foods, tastes, and recipes.

2.12 For each of the above vocabularies, consider writing a second XSL style sheet, this time not translating the XML to HTML but instead to a different markup language, such as WML, the markup language for WAP-enabled mobile telephones. Such a style sheet should be geared toward displaying the information on small mobile devices with limited bandwidth and limited screen space. You could use one of the freely available WAP simulators to display the results.