Describing and Manipulating XML Data

Sudarshan S. Chawathe
Department of Computer Science
University of Maryland
College Park, MD 20904
chaw@cs.umd.edu

Abstract

This paper presents a brief overview of data management using the Extensible Markup Language (XML). It presents the basics of XML and the DTDs used to constrain XML data, and describes metadata management using RDF. It also discusses how XML data is queried, referenced, and transformed using stylesheet language XSLT and referencing mechanisms XPath and XPointer.

1 Describing XML Data

The *Extensible Markup Language (XML)* [BPSM98] models data as a tree of *elements* that contain *character data* and have *attributes* composed of name-value pairs. For example, here is an XML representation of catalog information for a book:

```
<book>
   <title>The spy who came in from the cold</title>
   <author>John <lastname>Le Carre</lastname></author>
   <price currency="USD">5.59</price>
   <review><author>Ben</author>Perhaps one of the finest...</review>
   <review><author>Jerry</author>An intriguing tale of...</review>
   <bestseller authority="NY Times"/>
</book>
```

Text delimited by angle brackets (<...>) is *markup*, while the rest is *character data*. (Here, and in the rest of this paper, we introduce concepts informally as needed for our discussion; for formal specifications, see [W3C99].) Elements may contain a mix of character data and other elements; e.g., the book element contains the text "Here are some..." in addition to elements such as `title` and `price`. The element named `title` contains character data denoting the book title and is contained in the `book` element. Similarly, the element `price` contains character data denoting the book's price. This element also has an attribute named `currency` with value `USD`, represented using the syntax `attribute-name="attribute-value"` within the element's start-tag. In general, element names are not unique; e.g., the book element in our example contains two review elements. However, attribute names are unique within an element; e.g., the price element cannot have another attribute named currency. The syntax permits an empty element `<bestseller></bestseller>` to be represented more concisely as `<bestseller/>`. XML documents are called *well-formed* if they satisfy simple syntactic constraints, such as proper delimiting of element names and attributes and proper nesting of start and end tags.

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

1.1 DTD

As described above, XML provides a simple and general markup facility which is useful for data interchange. The simple tag-delimited structure of well-formed XML makes parsing extremely simple. However, applications that operate on XML data often need additional guarantees on the structure and content of such data. For example, a program that calculates the tax on the sale of a book may need to assume that each book element in its XML input includes a price subelement with a currency attribute and a numeric content. Such constraints on document structure can be expressed using a *Document Type Definition (DTD)*. A DTD defines a class of XML documents using a language that is essentially a context-free grammar with several restrictions. For example, one may use the following DTD declaration to constrain XML documents such as those in our book example:

```
<!ELEMENT book (title, author+, price, review*, bestseller?)>
<!ELEMENT title  (#PCDATA)>
<!ELEMENT author (#PCDATA|lastname|firstname|fullname)*>
<!ELEMENT price  (#PCDATA)>
<!ATTLIST price currency CDATA "USD"
                source   (list|regular|sale) list
                taxed CDATA #FIXED "yes">
<!ELEMENT bestseller EMPTY>
<!ATTLIST bestseller authority CDATA #REQUIRED>
```

The first line of this declaration is an *element type declaration* that constrains the contents of the book element. Following common convention, the declaration syntax uses commas for sequencing, parentheses for grouping, and the operators ?, *, and + to denote, respectively, zero or one, zero or more, and one or more occurrences of the preceding construct. Note that the declaration requires every book element to have a price sub-element. The second line declares the type for the title element to be *parsed character data* (implying an XML processor will parse the contents looking for markup). Note that the use of some element names (e.g., review, lastname) without a corresponding declaration is not an error; such elements are simply not constrained by this DTD. The last two lines declare bestseller to be an entity that must be empty and that must have an authority attribute of type character data. The declaration also indicates that the price element may have attributes currency, of type character data and default value USD; source, with one of the three values shown (an enumerated type) and default value list; and taxed, with the fixed value yes. The fixed attribute type is a special case of the default attribute type; it mandates that the specified default value not be changed by an XML document conforming to the DTD. Fixed-value attributes are convenient for ensuring that data critical to processing an element type is available with the desired value without requiring it to be explicitly specified for each element of that type. Our example DTD specifies that the book in our XML example must be taxed.

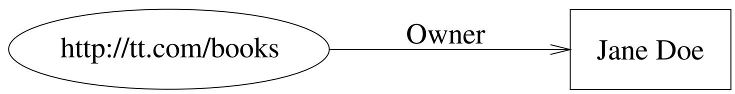
An XML document that satisfies the constraints of a DTD is said to be *valid* with respect to that DTD. The DTD associated with an XML document may be specified using several methods, one of which is the inclusion of a *document type declaration* <!DOCTYPE BOOKCATALOG SYSTEM "http://tt.com/bookcatalog.dtd">. in a special section near the beginning of a document, called its *prolog*. This declaration indicates that the XML document claims validity with respect to the BOOKCATALOG DTD which may be found at the indicated location.

The data modeling facilities provided by DTDs are insufficient for many applications. For example, we cannot use DTDs to require that the value of the element price be a fixed-precision real number in the range zero through 10000 with two digits after the point. Thus our tax-calculation application cannot rely on XML validity with respect to its DTD for such simple error-checking. The XML Schema proposal [BLM^{+}99, BM99] defines facilities that address these needs.

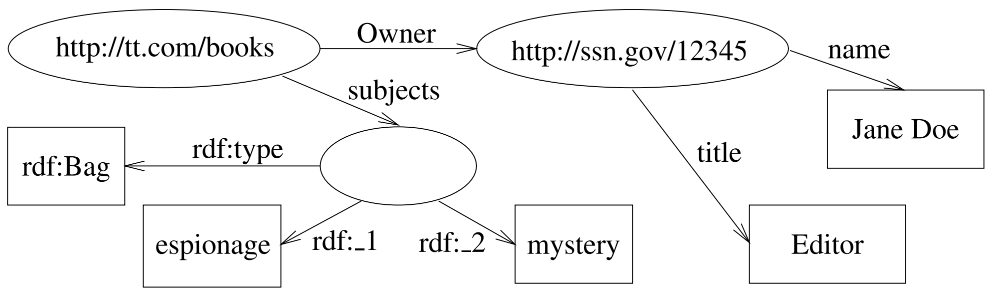
1.2 RDF

The *Resource Description Framework (RDF)* [LS99, BG99] provides a general method to describe metadata for XML documents. More specifically, RDF describes *resources*, which are objects (not necessarily Web-accessible) identified using *Uniform Resource Identifiers (URIs)* [BLFM98]. The attributes that are used to describe resources are called *properties*. RDF *statements* associate a property-value pair with a resource; they are thus triples composed of a *subject* (resource), a *predicate* (property), and an *object* (property value).

For example, suppose we associate the URI `http://tt.com/books` with the XML document in our `book` example above. We may indicate that the XML document is owned by "Jane Doe" using an RDF statement with the following triple: `(Subject: http://tt.com/books; Predicate:Owner; Object: "Jane Doe")` Such RDF statements are graphically represented using ovals for resources, rectangles for literal values, and directed arcs for properties:



The value of a property is not required to be a literal such as the string "Jane Doe" above; it may be another resource. For example, the following RDF graph indicates that the owner of the books data is the resource identified by URI `http://ssn.gov/12345`, which has the name Jane Doe and title Editor.



The above example also illustrates the RDF *container* facility. The subjects property of the books resource has the bag `{espionage, mystery}` as its value. RDF also provides container types sequence and alternative. Note that, like all RDF properties, the subjects property in our example has a single value (the bag). To make a statement about each member of a container, one must use a *distributive referent* attribute that intuitively modifies the meaning of the description element from a single statement to a container of statements (one for each element of the referenced container) [LS99].

RDF also specifies a concrete syntax based on XML for expressing RDF statements. For example, here is a complete XML document representing the above RDF graph:

```
<?xml version="1.0"?>
<rdf:RDF xmlns:rdf="http://w3.org/TR/1999/PR-rdf-syntax#"
         xmlns:bs="http://myschemas.org/books-schema#">
   <rdf:Description about="http://tt.com/books">
      <bs:Owner rdf:resource="http://ssn.gov/12345"/>
   </rdf:Description>
   <rdf:Description about="http://ssn.gov/12345">
      <bs:Name>Jane Doe</bs:Name>
      <bs:Title>Editor</bs:Title>
   </rdf:Description>
   <rdf:Description about="http://tt.com/books">
      <bs:Subjects>
         <rdf:Bag><rdf:li>espionage</rdf:li><rdf:li>mystery</rdf:li></rdf:Bag>
      </bs:Subjects>
   </rdf:Description>
</rdf:RDF>
```

This example also introduces some more XML concepts. Although technically not required, XML documents should begin with an *XML declaration*, similar to the one on the first line of our example, identifying the version of XML used. Element and attribute names appearing in an XML document may be qualified using *XML namespace declarations* [BHL99] such as those on lines 3–4 above. Our example introduces two namespaces. The first is identified using the URI http://w3.org/TR/1999/PR-rdf-syntax# and is assigned a shorthand `rdf`. This namespace contains the elements and attributes defined in [LS99]. The second is an imaginary *books schema* namespace containing properties that describe books (e.g., *Owner*); it is assigned the shorthand `bs`. Namespaces are an important addition to the base XML recommendation because they permit distributed, autonomous development of XML schemas without fear of name clashes. URIs are used in namespaces only for convenience in generating unique names and are not required to identify any Web resource.

RDF permits an intensional definition of bags using URIs. Such a definition is implicit in the use of a *distributive referent* of type `forEachPrefix`. For example, the `forEachPrefix` attribute below intensionally defines a bag containing all resources whose URIs have the specified prefix and establishes the Creator and Publisher of each resource in the bag.

```
<rdf:RDF xmlns:rdf="http://w3.org/TR/1999/PR-rdf-syntax#"
         xmlns:DC="http://purl.org/DC#">
   <rdf:Description aboutEachPrefix="http://cs.umd.edu/˜chaw">
      <DC:Creator>Sudarshan S. Chawathe</DC:Creator>
      <DC:Publisher>Dept. Computer Science, Univ. of Maryland</DC:Publisher>
   </rdf:Description>
<rdf:RDF/>
```

The above example uses terms from the *Dublin Core* content description model which is described at the URI shown. This model predates RDF and the RDF Schema recommendation [BG99] includes a schema for it.

One often needs to make statements about other statements (e.g., “Foo believes that the creator of Bar is Baz”). For this purpose, RDF allows a statement to be *reified*: It can be transformed into a resource of type `statement`, with properties `subject`, `predicate`, and `object`, to which additional properties (e.g., `authority` and `PGP-signature` below) may be attached:

```
<rdf:Description>
   <rdf:subject>Bar</rdf:subject>
   <rdf:predicate resource="http://purl.org/DC#Creator"/>
   <rdf:object>Baz</rdf:object>
   <rdf:type resource="http://w3.org/TR/1999/PR-rdf-syntax#Statement"/>
   <authority>Foo</authority>
   <PGP-signature>XmdkA093cDks...</PGP-signature>
</rdf:Description>
```

2 Manipulating XML Data

Given an XML document, one often needs to transform it to better suit the needs of an application. For instance, we may wish to generate a printed catalog containing information about all the books in our running example. In one printed catalog, we may wish to include only the title, authors, and price of each book, skipping other details such as reviews. We may also wish to generate a smaller catalog containing only bestsellers. Further, we may wish to automatically generate a table of contents for these catalogs. Of course, one can implement such applications by writing procedural programs that access the required parts of the source XML document, perhaps using a convenient object interface such as the *Document Object Model (DOM)* [$\mathrm{A}^{+}98$]. However, XML applications, like database applications, stand to benefit from a declarative languages. Note that the languages described in this section, like RDF in the previous section, operate on the logical tree structure of an XML document (e.g., as supported by DOM), not on its serialization syntax.

2.1 XSL

The *Extensible Stylesheet Language (XSL)* is a language for transforming and formatting XML. Recently, the transformation and formatting parts of XSL were separated. In this paper, we focus on the *XSL transformation language*, called *XSLT* [Cla99], and the related *XPath* [CD99] and *XPointer* proposals [DJ99].

An XSLT stylesheet is a collection of *transformation rules* that operate (non-destructively) on a source XML document (source tree) to produce a new XML document (result tree). Each rule consists of a *pattern* and a *template*. During rule processing, patterns are matched against the nodes of the source tree, and the template is instantiated (typically using references to the matched nodes) to produce part of the result tree. Templates may contain, in addition to literals and references to matched nodes, explicit instructions for creating result tree fragments. Rule processing starts by instantiating the template of the rule that matches the root element of the source tree. (XSLT uses a conflict resolution mechanism when several rules match a node and default rules when no rules match a node.) Additional elements are processed only when they have been selected for processing by the template of some previously processed element.

Here is an XSL stylesheet for transforming an XML document containing book elements (from our running example) to an XHTML [XHT99] document that pretty-prints the title, author, and price of each book, and that includes only the first review for each book. (XHTML is a reformulation of HTML 4.0 in XML.)

```
                    <xsl:stylesheet xmlns:xsl="http://w3.org/XSL/Transform/1.0"
                                    xmlns="http://w3.org/TR/xhtml1"
                                    indent-result="yes">
<!-- Rule 1 --> <xsl:template match="/">
                       <html><head><title>Our New Catalog</title></head>
                          <body>
                             <xsl:apply-templates/>
                          </body>
                       </html>
                    </xsl:template>
<!-- Rule 2 --> <xsl:template match="book/title">
                       <h1><xsl:apply-templates/></h1>
                    </xsl:template>
<!-- Rule 3 --> <xsl:template match="book/author">
                       <b><xsl:apply-templates/></b>
                    </xsl:template>
<!-- Rule 4 --> <xsl:template match="book/price">
                       <xsl:apply-templates/> <xsl:apply-templates select="@*">
                    </xsl:template>
<!-- Rule 5 --> <xsl:template match="book/review[1]" priority="1.0">
                       <xsl:apply-templates/>
                    </xsl:template>
<!-- Rule 6 --> <xsl:template match="book/review" priority="0.5">
                    </xsl:template>
                    </xsl:stylesheet>
```

The first three lines declare the XSL and XHTML namespaces used by the stylesheet. The XHTML namespace is made the default namespace (by skipping the local shorthand in the declaration); thus, unqualified element and attribute names (e.g., head) are implicitly in the XHTML namespace. (In XML, text between the *comment delimiters* <!-- and --> is ignored by processors.) Each template element describes one transformation rule. The match attribute of a template element specifies the rule pattern while its content is the template used to produce the corresponding portion of the result tree. The pattern "/" of the first rule denotes the root of the source tree. The template contains some standard XHTML header and trailer constructs. The apply-templates element is a rule-processing instruction that denotes recursive processing of the contents of the matched element. (XSLT includes several other instructions which permit templates with constructs such as for-loops, conditional sections, and sorting.) The second rule's pattern, "book/title" matches a title element if its parent is a book element. The template calls for recursive processing of the contents, enclosed in XHTML literals for bold display (...). XSL processing includes implicit rules that match elements, attributes, and character data (text) not matched by any explicit rules; these rules simply copy data from source to result tree. In our example, all character data (such as the the text "The spy..." in the title) is copied to the result tree. Rule 4, for processing price elements, is similar but includes an additional apply-template instruction to extract the currency attribute using the syntax @*. Rule 5 matches only the first review element in each book element due to the "[1]" specification. The template simply copies the contents to the result tree (using recursive processing with apply-templates combined with the default rules). We ensure that the first review for each book is processed using Rule 5 instead of Rule 6 by assigning Rule 5 a higher priority.

2.2 XPath

XSLT rules contain patterns that are matched against nodes (elements, attributes, etc.) in the XML source tree. The language for specifying these patterns is *XML Path Language (XPath)* [CD99]. Principally, XPath defines the syntax and semantics of *path expressions* such as the following, which matches the last report child (in document order) of the weather descendants of the node with unique identifier "favorites":

```
id("favorites")/descendant::weather/child::report[position()=last()]
```

Path expressions are evaluated in a *context* consisting of a node called the *context node*, a set of nodes called the *context node list*, a set of variable bindings, a function library, and the set of namespaces in scope. Path expressions may be *relative*, selecting nodes by navigating from the current context node, or *absolute*, selecting nodes by navigating from the document root. A path expression consists of a sequence of `/`-separated *steps*, where a step is a *basis* followed by an optional list of *predicates*. Informally, a basis indicates a navigational selection of nodes based on the current context, while the predicate list narrows the list of selected nodes using properties such as position and value. A basis is of the form *AxisName::NodeTest*, where *AxisName* refers to one of several inter-node relationship types and *NodeTest* is a selection condition based on this relationship.

Our path expression example above has three steps: (i)a predefined function that selects the (unique) node that has an `ID` attribute of value "`favorites`"; (ii) a basis `descendant` and node test `weather` that returns a list, in document order, of all `weather` descendants of the context node; and (iii) navigation from these `weather` nodes, giving a list of their `report` children, which are filtered using the predicate in square brackets to yield only the last `report` child for each `weather` node (in document order). The function `position` returns the position of the current context node (at evaluation time) in the context node list, while `last` returns the number of nodes in this list. These functions are from the XPath *core function library*, which includes other functions that return properties of the context node and list as well as common utility functions on numbers, strings, and booleans. Note that the result of performing a basis step is a list of context nodes, not a set. The list order depends on the axis. Intuitively, the basis nodes are in ascending order of distance from the context node.

In addition to `child` and `descendant`, XPath provides the following axes. The `parent` axis contains the parent, if any, of the context node; the parent of an attribute or namespace node is defined to be the element it modifies. The `following-sibling` axis contains siblings of the context node that precede it in the document. The `following` axis contains only element nodes that strictly follow the context node in the document. Descendants of the context node are excluded. The `preceding` axis is analogous; it contains element nodes that strictly precede the context node. The `ancestor` axis contains the proper ancestors of the context node (based on the `parent` axis). The `attribute` (`namespace`) axis contains the attribute (respectively, namespace) nodes attached to the context node if it is an element node, and is empty otherwise. Finally, the `ancestor-or-self` and `descendant-or-self` axes are defined as their names suggest.

Node tests may also use constraints on attribute nodes. For our books example, the following XPath selects book nodes whose price child has attribute currency equal to `USD` and attribute source equal to

```
list: root()/descendant::book/child:price[attribute::currency="USD"
                 and attribute::source="list"]/parent::node()
```

This syntax for path expressions is verbose. XPath also defines an *abbreviated syntax* by mapping it to this syntax. For example, `./foo` and `.//foo` select all `foo` children and descendants, respectively, of the context node; `./foo[3]` selects the third `foo` child of the context node. Further, "`.`" and "`..`" are abbreviations for `self::node()` and `parent::node()`, respectively. Thus, we may rewrite our two examples as:
`id("favorites")//weather/report[last()]` and `//book/price[@currency="USD" and @source="list"]/..`

2.3 XPointer

Applications may need to address precise portions within XML documents that cannot be modified, e.g., an XML tutorial may wish to annotate specific sections, paragraphs, or sentences of the XML recommendation [BPSM98] without modifying it. (This application is described in [Bra98].) Addressing parts of XML documents is also important when transforming XML using XSLT and XPath as described above. However, the addressing capabilities of XPath are not sufficient. For example, the above application may wish to highlight a region of the XML recommendation that is not a well-formed XML fragment. The *XML Pointer Language (XPointer)* [DJ99] extends XPath to support such applications by adding two new axes to specify basis steps in XPath.

The *range axis* addresses the XML region bounded by the locations addressed by its two arguments. For example, the following XPointer selects the document region between the first and fifth book reviews (inclusive) for our running example:

```
//book/range::review[1],following-sibling::review[4]
```

The range axis is extremely useful for denoting all regions that are marked using a pair of empty elements such as `my-edits-begin/-end` in the following example:

```
XML fragment: ...<Observations>
                     <Temp> 98 99 101 92 <my-edits-begin/> 76 32 99 </Temp>
                     <Pressure> 30 31 33 32 </Pressure>
                  </Observations>
                  <Conclusion>Interestingly, <my-edits-end/>...
XPointer:     //range::descendant::my-edits-begin,following::my-edits-end[1]
```

This XPointer specifies a range beginning at each `my-edits-begin` element and ending at the next `my-edits-begin` element. It illustrates two restrictions on the range axis: (i) although its first argument may reference multiple locations, each such location must be followed by exactly one location referenced by the second argument; (ii) unlike other axes, a range axis may denote regions that cannot be mapped to well-formed context node lists (e.g., the region between the edit markers above). Hence, range axis results cannot be processed further using the XPointer mechanism. (They are intended for use by application programs.)

The *string* axis selects regions using character-based matches (in contrast with the node-based matches used by other axes). The expression "`string::`*n*,*M*,*p*,*l*," where *n*, *p*, and *l* are integers and *M* is a string, selects the sequence of *l* characters starting at the *p*'th position following the last character of the *n*'th occurrence of the pattern *M*. For our running example, the following selects the tenth occurrence of "spy" within a book review element or its descendants, along with 20 characters before and after the word: `/book/review//string::10,"spy",23,43`.

XPointer also adds two functions that specify absolute location paths (similar to the `/` and `id()` expressions used by XPath). The `here()` function locates the element that directly contains (as content or attribute) the XPointer itself (instead of a node in the source tree). The absence of such an element is an error. The `origin()` function is intended for link-traversal and refers to the resource from which the traversal in context began, the absence of such a traversal signaling an error.

References

[A+98] V. Apparao et al. Document Object Model (DOM) level 1 specification version 1.0. W3C Recommendation, October 1998. Available at `http://www.w3.org/TR/REC-DOM-Level-1-19981001`.

[BG99] D. Brickley and R. Guha. Resource Description Framework (RDF) schema specification. W3C Proposed Recommendation, March 1999. Available at `http://www.w3.org/TR/PR-rdf-schema`.

[BHL99] T. Bray, D. Hollander, and A. Layman. Namespaces in XML. World Wide Web Consortium Recommendation. Available at `http://www.w3.org/TR/REC-xml-names`, January 1999.

[BLFM98] T. Berners-Lee, R. Fielding, and L. Masinter. IETF (Internet Engineering Task Force) RFC 2396: Uniform Resource Identifiers (URI): Generic syntax, August 1998. Available at `http://www.ietf.org/`.

[BLM+99] D. Beech, S. Lawrence, M. Maloney, N. Mendelsohn, and H. Thompson. XML schema part 1: Structures. W3C Working Draft, May 1999. Available at `http://www.w3.org/TR/1999/xmlschema-1/`.

[BM99] P. Biron and A. Malhotra. XML schema part 2: Datatypes. W3C Working Draft, May 1999. Available at `http://www.w3.org/TR/1999/xmlschema-2/`.

[BPSM98] T. Bray, J. Paoli, and C. Sperberg-McQueen. Extensible markup language (XML) 1.0. World Wide Web Consortium Recommendation. Available at `http://www.w3.org/TR/REC-xml`, February 1998.

[Bra98] T. Bray. Using XML to build the annotated XML specification, September 1998. Available at `http://www.xml.com/pub/98/09/exexegesis-0.html`.

[CD99] J. Clark and S. DeRose. XML path language (XPath) version 1.0. W3C Working Draft, July 1999. Available at `http://www.w3.org/TR/WD-xpath-19990709`.

[Cla99] J. Clark. XSL transformations (XSLT) version 1.0. W3C Working Draft, July 1999. Available at `http://www.w3.org/TR/WD-xslt-19990709`.

[DJ99] S. DeRose and R. Janiel Jr. XML pointer language (XPointer). W3C Working Draft, July 1999. Available at `http://www.w3.org/TR/WD-xptr-19990709`.

[LS99] O. Lassila and R. Swick. Resource Description Framework (RDF) model and syntax specification. W3C Proposed Recommendation, January 1999. Available at `http://www.w3.org/TR/PR-rdf-syntax`.

[W3C99] The World-Wide Web Consortium. `http://www.w3.org/`, 1999.

[XHT99] XHTML 1.0: The extensible hypertext markup language. W3C Working Draft, May 1999. Available at `http://www.w3.org/TR/1999/xhtml1-19990505/`.