Introduction to XML

Patrick Yott

SUMMARY. This chapter begins with a brief history of markup technologies and an examination of two of the most commonly encountered markup languages: HTML and XML. The basic structural components of an XML document are described, and the rules for creating “well-formed” documents are discussed. The concept of data modeling and document “validity” will be demonstrated using a simple DTD (Document Type Definition), and several markup examples will follow. The notions of XML as a data interchange system and XSLT as a transformation and display language will be examined. [Article copies available for a fee from The Haworth Document Delivery Service: 1-800-HAWORTH. E-mail address: docdelivery@haworthpress.com] Website: [http://www.HaworthPress.com] © 2005 by The Haworth Press, Inc. All rights reserved.

KEYWORDS. Markup languages, XML, data modeling, document validity, DTD (Document Type Definition)

A BRIEF HISTORY OF MARKUP LANGUAGES

The past 35 years have seen a movement from specific encoding of electronic documents to generic encoding of electronic documents. In specific encoding an electronic document contains control codes that
would dictate how the document was to be formatted. These codes were generally specific to one hardware or software configuration and would not work if one moved from, for example, Waterloo script on a mainframe to WordPerfect on a DOS machine. Generic encoding, which many trace back to Tunnicliffe’s 1967 presentation on separating content from format (Goldfarb 1990, 567), focuses on describing the structural and semantic content of a document, and leaves formatting instructions to external documents and tools thus allowing content to be shared between computing environments.

By 1969, engineers at IBM had taken Tunnicliffe’s ideas and produced a language called Generalized Markup Language, or GML. In 1986, the International Standards Organization (ISO) adopted an advanced version of GML, the Standard Generalized Markup Language (SGML) as an ISO standard. SGML was a meta-language—a set of rules and principles with which one could create content specific tag sets, such as the Text Encoding Initiative (TEI), the Encoded Archival Description (EAD), and in 1991, the Hypertext Markup Language (HTML).

In 1996, faced with the successful implementation of HTML across the World Wide Web, and a simultaneous dissatisfaction with the limitations of the finite HTML tag set and its focus on presentation (see below), the World Wide Web Consortium adopted XML as a specification. XML fits the middle ground between SGML and HTML. Like SGML it is a meta-language and has led to the rapid development of content specific markup languages, including X versions of the TEI, EAD, and HTML languages. Because it is less complex (and leads to more easily processed documents) than SGML, a wide variety of open-source and commercial XML applications have quickly followed its introduction.

From the perspective of the librarian or information manager, the widespread development of XML technologies has had profound effects. Where MARC once ruled as the primary metadata format used by librarians and library systems, librarians are routinely encountering XML based metadata standards, such as the TEI (full text databases), EAD (manuscript and archival finding aids), METS (complex digital object structures), and MODS/Dublin Core (alternative descriptive cataloging systems).

**XML, HTML, WHAT’S THE DIFFERENCE?**

Before we go any further, we must eliminate the most common of misconceptions—XML is not HTML on steroids (University of Virginia
The most fundamental difference between HTML and XML is that XML is a set of rules that are used to create markup languages while HTML is itself a markup language. XML is a meta-language. While one can correctly state that she is marking up a document in HTML, it would be technically inaccurate to state that she is marking up a document in XML. It would be better to state that she is marking up a document in the TEI or EAD markup language, both of which are implementations of XML.

HTML is a remarkably simple markup language. It was this simplicity that allowed the World Wide Web to grow at the bewildering rate that it has. Quickly learned and easily implemented (there is a wide variety of HTML editors available), HTML contains a finite set of tags that are used to instruct browsers how to display information and how to link between documents, images, and other objects present on the web.

So what then is XML?

On one level, XML is a protocol for containing and managing information. On another level, it’s a family of technologies that can do everything from formatting documents to filtering data. And on the highest level, it’s a philosophy for information handling that seeks maximum usefulness and flexibility for data by refining it to its purest and most structured form. (Ray 2001, 2)

It is, in essence, a methodology for creating self-describing documents.

Looking at this difference in less abstract terms, one would use HTML to describe the appearance of a document and would use an XML language to describe the structure of a document. For example, suppose you were cataloging a collection of musical recordings and wanted to move beyond MARC to handle this task. For this example, and throughout this article, we’ll look at creating a non-MARC catalog record for Fripp and Eno’s groundbreaking recording: *No Pussyfooting* (EG, 1973). Let’s look at how this could be done with HTML and a portion of an implementation of XML.

```
<html>
<body>
<h2 align="center">No Pussyfooting</h2>
<strong>Fripp and Eno</strong>
<br>
```
Both of these documents contain identical information, but differ greatly in the utility of that information. If we were principally concerned with display, the HTML example would work very well, and one might ultimately want to display the XML content in such a manner. But if we wanted, for example, to identify specific catalog records by searching for a given label, the XML-based representation would be
more effective. In the HTML example, the record label is visually identifiable but is not encoded in a way that would permit efficient machine processing.

RULES FOR WELL-FORMEDNESS

When working with documents encoded using an XML-based markup language we must be concerned with both the shape of the document (its well-formedness) and the syntax of the language used (its validity). We’ll deal with validity in a later section. Before we can do anything with a document based on an XML markup language, that document must be well-formed. There are four basic tenets of well-formedness: (1) every opening tag must be matched with a corresponding closing tag; (2) tags must nest cleanly; (3) values assigned to attributes must be placed within quotation marks; and (4) every document must have a root element. Let’s take a quick look at each of these basic rules.

Opening and Closing Tags

Look at our HTML example above. In the list of performers each performer is encoded as a list item (the \texttt{<li>} element in HTML), and are encoded with the open (\texttt{<li>}) tag but without the close (\texttt{</li>}) tag. Since HTML is derived from SGML and since SGML allows for tag minimization (the lack of closing tags), this is not a problem, and web browsers have been developed to allow for the resulting ambiguity. In an XML-based markup language (such as XHTML), we would have to both open and close our list item tags, like this: \texttt{<li>Ritchie, Brian (Guitar Bass)</li>}. We have now removed any ambiguity as to where one list item ends and another begins.

Case counts. In an XML-based markup language \texttt{<Name>} is a different element from \texttt{<name>}. While they may look confusingly similar to human eyes, any software that can process an XML-based structure can easily differentiate the two. In HTML \texttt{<h2>} and \texttt{<H2>} are considered synonymous, so you can open a node with \texttt{<h2>} and close it with \texttt{</H2>} without generating an error. Not so in XML—these two tags are considered to be completely unrelated.

For elements that don’t contain any data or other elements (empty elements), these tags can close themselves either by opening and immediately closing or by closing within their opening tag.
Nesting Tags

It is important that when you open a tag within a previously opened tag, that you close the inner tag pair before closing the outer pair. While Netscape browsers used to consider improperly nested tags an error (they no longer do), HTML will render when a document is so encoded. In a document based on XML markup rules, any document that violates this practice will fail to parse and will generate an error.

Attributes and Values

Attributes are used to refine an element. In our XML example above, we refine each performer node (that area bounded by a tag pair) by indicating; in an attribute, what the role of each performer was in this recording. The value assigned to each role attribute must be enclosed within matching quotation marks (either single/single or double/double). Not to continually pick on HTML, but in HTML while this is considered good practice, you are not required to so enclose your values.

The Root Element

Every well-formed XML document contains a root element. The root element opens and closes the document and can only occur once in the document.

TESTING FOR WELL-FORMEDNESS

The rules for well-formedness ensure that any simple XML-enabled tool, such as an up-to-date web browser, can read your document. In fact, “reading” your XML document in a browser is often the simplest and quickest test for well-formedness. If the document is well-formed, it will parse, and can be displayed.

When a web browser parses a document, it does so in complete ignorance of the semantic and syntactic rules the lay behind the tags it encounters. We call this type of parser a non-validating parser. Because the browser is limited to checking that the fundamental rules of any XML document are followed, it is a very useful tool when encoding a document that doesn’t follow any specific document model (more on this soon . . .). Notice the plus and minus signs to the left of your tags (see Figure 1). These allow you to expand and collapse any node within
your document, and are a very useful way of exploring a document’s structure.

If a document is not well-formed, a non-validating parser will stop processing (displaying if you are using a web browser) the document when it encounters its first well-formedness error. It generally provides an error message alerting you to the nature of the violation (see Figure 2). For example, the following example displays the Internet Explorer error message we generate if we attempt to close a creator tag with a credit tag, like this:

```xml
<creator>Brian Eno</credit>
```

**DOCUMENT MODELING AND DOCUMENT VALIDITY**

The rules described above are obviously ignorant of content considerations—they only insure that a document adheres to a minimal syntactical cleanliness. In order to have documents that maintain a predictable structure and that can be shared between various organizations and sys-
tems, a content model needs to be established and standardized. Documents that conform to this content model are said to be valid.

Content models have traditionally been recorded in documents called Document Type Definitions (DTDs), a document modeling technology dating back to the advent of GML and SGML. A newer technology, the XML Schema, encodes a document model as an XML document and offers more specific data modeling options than the DTD. For the purposes of this introduction we’ll explore the DTD as a document modeling language.

Found within a DTD (or schema) are the grammar and vocabulary for encoding a given type of document—the DTD lists all the elements that are permissible, the order and/or places where those elements can exist within your document and any specific nesting rules. Your document is said to be valid when it adheres to these rules. If you use an element that is not listed in the DTD, or use a listed element in an unacceptable way, your document will not validate. Only when members of a community agree to encode their documents according to a specific content model does it becomes possible to share content. (A registry of published
DTDs and schema is available from xml.org at http://www.xml.org/xml/registry.jsp. The following DTD was used to create the XML record seen earlier.

```xml
<!-- Sample DTD for encoding musical recordings -->
<!-- Patrick M. Yott, July 2004 -->
<!ELEMENT recording (albuminfo, tracklist)>
<!ATTLIST recording
  ID ID #REQUIRED
  condition (mint | fine | good | poor) #REQUIRED
>
<!ELEMENT albuminfo (title, creator*, label, genre+, year, credits, time)>
<!ELEMENT creator (#PCDATA)>
<!ELEMENT credit (creator, role+)>
<!ELEMENT credits (credit+)>
<!ELEMENT label (#PCDATA)>
<!ATTLIST label
catno CDATA #IMPLIED
>
<!ELEMENT genre (#PCDATA)>
<!ELEMENT year (#PCDATA)>
<!ELEMENT role (#PCDATA)>
<!ELEMENT time (#PCDATA)>
<!ELEMENT title (#PCDATA)>
<!ELEMENT track (title, credits, time)>
<!ATTLIST track
  sequence CDATA #REQUIRED
>
<!ELEMENT tracklist (track*)>
```

In looking at this DTD, you should notice that the structure is deceptively simple—it consists of a series of element and attribute declarations. This is the vocabulary for your document. Within each declaration are the rules for constructing that element or attribute. This is the grammar for your document. Let’s look at these in more detail.

**Declaring an Element**

Elements are declared with this basic syntax:

```xml
<!ELEMENT name content-model>
```
The name of an element represents a tag that you are permitted to use in your XML document. The content model dictates what that element can contain in your document. There are five general content models that are used with elements: empty, parsed character data (#PCDATA), element, mixed, and all. Let’s look at each of these in turn.

Empty

An empty content model specifies that there can be nothing between the opening and closing of the element in your document. In fact, elements designed as empty elements generally “self-close” when used, such as \(<\text{hr}/>\) or \(<\text{br}/>\). An empty element may contain attributes.

Parsed Character Data

PCDATA content models specify that you can include text between the open and close of the tag set (referred to as a node), but that you cannot include any other tags. These are declared as:

\[\text{<!ELEMENT title (#PCDATA)>}\]

Element

Element data models signify that the element can contain other elements, and they specify the order and quantity of these nested elements. For example, the declaration:

\[\text{<!ELEMENT recording (albuminfo, tracklist)>}\]

specifies that any recording node in your document must include an album info node followed by a tracklist node.

The order in which elements are listed in this content model specifies the order in which they must be used in your document. The comma equates to “followed by,” so in the example above, albuminfo must be followed by tracklist and both occur within recording. If two elements in a content model are separated by a vertical bar (|), this specifies that you can use one or the other but not both.

Aside for the order of elements in the content model, the DTD also specifies the quantity of each element. It does this by affixing a symbol (*, +, ?) to the element name.
Thus, the element declaration:

```xml
<!ELEMENT track (title, credits, time)>
```

specifies that if you have a track node, it must contain one title node, followed by one credits node, followed by one time node. Notice that this declaration doesn’t provide any information about when and where you can use a track node. You have to examine all the other element declarations in the DTD to determine where this element is acceptable. In this case, you’ll find it listed within the tracklist element’s content model, which is itself found within the recording content model.

**Mixed**

A mixed content data model allows for both PCDATA as well as elements to be included within a given element node. The same rules that govern order and quantity for element data models apply to the mixed data model. Although our DTD doesn’t contain any mixed content models, the general structure would look like this:

```xml
<!ELEMENT p (#PCDATA | i | br)*>
```

This declares that you can use an element named p (the paragraph tag) and it can contain a mix of character data, italic tags (<i>), and line break tags (<br>).

**All**

The all data model specifies that there is no restriction whatsoever on what a particular element can contain, and as such is not very useful since there is really no point in declaring an element if it doesn’t have any structural/content restrictions.

**Declaring an Attribute**

The attributes for any given element are declared using this basic syntax:
In our DTD, the label element can take one attribute, the catno (catalog number) attribute. This is modeled in the DTD as:

```xml
<!ATTLIST label
    catno CDATA #IMPLIED
>
```

You can list multiple attributes for any given element. Our DTD defines two attributes for the recording element using this syntax:

```xml
<!ATTLIST recording
    ID ID #REQUIRED
    condition (mint | fine | good | poor) #REQUIRED
>
```

Note that the first attribute is named ID and is of type ID. There are several types of attributes available in a DTD. The type determines what type of content can be assigned as a value to a given attribute. The three most commonly used types of attribute are character data (CDATA), an enumerated list of values, or a specific (tokenized) constraint, such as uniqueness.

**CDATA**

A CDATA attribute type signifies that the value can be any text string. The label element can have an attribute and value of catno="2-23845", for example. CDATA types allow us to use any text string as a value, including numbers. In our DTD this is controlled by:

```xml
<!ATTLIST label
    catno CDATA #IMPLIED
>
```

**Enumerated List**

The recording element accepts an attribute called condition (in addition to the ID attribute), but instead of letting the encoder enter any value,
restricts valid values to a standardized list. This attribute is controlled like this:

```xml
<!ATTLIST recording
    ID ID #REQUIRED
    condition (mint | fine | good | poor) #REQUIRED
>
```

If the individual encoding a document enters a value other than mint, fine, good, or poor, the document may be well-formed (assuming she has followed good form throughout), but it will not be valid.

**Tokenized Attributes**

A tokenized attribute has some sort of constraint placed upon it. There are seven types of constraints, but for now we’ll look at the ID and IDREF/IDREFS tokenized attribute types.

The ID data type is one of the more useful tokenized attributes because its value must be unique. No two elements can have the same value assigned to an ID type attribute. This ID is useful anytime you want to “reach” into your document to retrieve a specific piece of information. For instance, if you have a bunch of albums in your database, it might be a good idea to give them an ID attribute so that each album can be referenced uniquely. The recording element in our DTD has an attribute of type ID:

```xml
<!ATTLIST recording id ID #REQUIRED>
```

In this line, the name of the attribute is id (in lower case) and the type is ID (which is always in upper case).

The IDREF data type is used to reference an ID data type that has been specified at some other point in the document. (The only difference between IDREF and IDREFS is that IDREFS point to/refer to more than one specified ID. In an IDREFS data type, a whitespace character separates each referenced ID.)

In addition to assigning a type to an attribute (thus limiting the types of data that can be assigned to it), the DTD also provides behavioral guidance on the use of the Attribute. The two most commonly encountered behaviors are #REQUIRED and #IMPLIED.
**Required**

When an attribute is declared with the #REQUIRED behavior, you are required to use that attribute. Failure to include that attribute whenever you use the element it is assigned to will cause the document to fail a validity test. So, the attribute declaration:

```xml
<!ATTLIST recording
    ID ID #REQUIRED
>
```

informs us that we are required to use an attribute called ID (which is of type ID) whenever we open a recording node.

**Implied**

An implied attribute indicates that you can decide whether or not to include this attribute in your document. This declaration from our DTD:

```xml
<!ATTLIST label
    catno CDATA #IMPLIED
>
```

informs us that we can choose to include (or not to include) the catno attribute when we use the label element. Thus, a document will be valid with or without this attribute assigned to the label element. However, if we do choose to include this attribute, it must have unique values, as it is of type ID.

**TESTING FOR DOCUMENT VALIDITY**

As we saw earlier, a document must be well-formed before it can be processed by any XML software. A second, more rigorous test involves checking an encoding against a document model (DTD or schema) to make sure all elements and attributes are used appropriately. To do so, we need a validating parser. There are several validating parsers available (for reviews and links to XML software, check out the XML Cover Pages at http://xml.coverpages.org/). In the following examples, we will perform validity tests using the xmlint parser from Microsoft which
we have bundled into the NoteTab Light text editor. (The NoteTab Light software used in this article can be downloaded from http://notetab.com. The clips and batch files, along with installation notes, can be downloaded from http://dl.lib.brown.edu/staff/yott/software/xmltool.html.)

We’ll be stepping through the encoding of this document in a bit, but let’s assume we have the following document to validate (see Figure 3). For the sake of this example, we have encoded the condition as “excellent,” which is a violation of the content model for the condition attribute. The DTD we examined earlier only accepts one of four possible values (mint, fine, good, or poor), so while this remains well-formed, it is no longer valid.

If we attempt to validate the document, we get the error message shown in Figure 4.

If we correct the error by replacing the condition value with “fine,” and attempt to validate the document again, we receive the name of the validated document with no errors reported (see Figure 5). This informs us that the document is now both well-formed and valid.

FIGURE 3. An XML Document in NoteTab Light
FIGURE 4. Error Message Generated by xmlint

Attribute 'condition' has an invalid value according to the DTD/Schema.
URL: file:///C:/xmlfiles/docs/cd.xml
Line 00003: <recording id="01" condition="excellent">
Pos 00041: ------------------------------------------

FIGURE 5. xmlint Message from Successful Validation
CREATING AN XML DOCUMENT

In this section, we’ll step through the creation of an XML document using the DTD explored earlier.

Encode the Prologue

The Prologue exists at the top of your document, and contains very important information that will be used by your operating system and any XML aware applications you use to further process or analyze this document. The prologue begins with a processing instruction that tells your operating system that the document is an XML document followed by information identifying the root element and the DTD that governs the document.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE recording SYSTEM "c:\dtds\music.dtd">
```

The Root Element

In our prologue, we indicated that recording was the root element of our document. If you examine the DTD, you will notice that recording is the only element that doesn’t exist within any other element and can only occur once. Usually, but not always, you’ll find the root element listed at the top of the DTD, but the placement of any element within the DTD is purely arbitrary.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE recording SYSTEM "c:\dtds\music.dtd">
<recording>
<recording>
```

Our recording node is not correct. Why? Look back at the DTD; the recording element has two required attributes—the ID attribute, which is of type ID, and the condition attribute, which must have a value from the specified list. So a correctly encoded root element would look like this:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE recording SYSTEM "c:\dtds\music.dtd">
<recording ID="cd1" condition="fine">
<recording>
```
Note that while you indicated all needed attributes when you open a tag, you don’t list them when you close it. This is important—if you close a node and list the attributes within the closing tag your document will not be well-formed.

When you try to parse this, you get a new error message stating that the recording element cannot be empty. Look at the DTD and examine the content model for this element. Note that it specifies two elements must be placed within the recording node. Both albuminfo and tracklist must occur, once, and in that order.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE recording SYSTEM "c:\xmlfiles\dtds\music.dtd">
<recording ID="cd1" condition="fine">
  <albuminfo>
  </albuminfo>
  <tracklist>
  </tracklist>
</recording>
```

When you try to validate the document now, you receive a similar error message regarding the albuminfo element. Look again at the DTD—what is the content model for albuminfo? Follow this process for each element you place within the albuminfo node. Below is an abbreviated encoding, skipping some of the creators for brevity.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE recording SYSTEM "c:\xmlfiles\dtds\music.dtd">
<recording ID="cd1" condition="fine">
  <albuminfo>
    <title>No Pussyfooting</title>
    <creator>Fripp and Eno</creator>
    <label catno="2">EG</label>
    <genre>Experimental</genre>
    <genre>Electronic</genre>
    <year>1973</year>
    <credits>
      <credit>
        <creator>Brian Eno</creator>
        <role>Synthesizer</role>
        <role>Keyboards</role>
      </credit>
    </credits>
  </albuminfo>
</recording>
```
The last section of the document to encode is the tracklist node. Looking at the DTD, we notice that the content model of tracklist specifies that it includes a minimum of one track element, and only the track element can exist within it. Next, examine its content model. Each track that you include in the tracklist node contains a title node, a credits node, and a time node. Each of these nodes can only occur once, and in the order specified.

Once the tracks are encoded (there are only two tracks on this particular CD), the complete XML document looks like this:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE recording SYSTEM "c:\xmlfiles\dtds\music.dtd">
<recording ID="cd1" condition="fine">
  <albuminfo>
    <title>No Pussyfooting</title>
    <creator>Fripp and Eno</creator>
    <label catno="2">EG</label>
    <genre>Experimental</genre>
    <year>1973</year>
  </albuminfo>
  <tracklist>
    <track><title>Part II: How to Create, Apply, and Use Metadata 231</title></track>
  </tracklist>
</recording>
```
<credit>
<creator>Brian Eno</creator>
<role>Synthesizer</role>
<role>Keyboards</role>
<role>Vocals</role>
<role>Producer</role>
<role>Performer</role>
<role>Treatments</role>
<role>VCS 3 Synthesizer</role>
</credit>

<credit>
<creator>Robert Fripp</creator>
<role>Guitar</role>
<role>Producer</role>
<role>Remastering</role>
</credit>

<time>39:38</time>
</albuminfo>

<tracklist>
<track sequence="1">
<title>The Heavenly Music Corporation</title>
<credits>
<credit>
<creator>Brian Eno</creator>
<role>Composer</role>
</credit>
<credit>
<creator>Robert Fripp</creator>
<role>Composer</role>
</credit>
<time>20:55</time>
</track>
<track sequence="2">
<title>Swastika Girls</title>
<credits>
<credit>
<creator>Brian Eno</creator>
<role>Composer</role>
</credit>
<credit>
<creator>Robert Fripp</creator>
<role>Composer</role>
</credit>
<time>20:55</time>
</track>
</tracklist>
MY XML IS WELL-FORMED AND VALID, NOW WHAT?

One of the primary advantages XML holds over HTML is that it allows you to create whatever elements and tags you need to achieve a particular goal. One of the primary complications of XML when contrasted to HTML is that it allows you to create whatever elements and tags you need to achieve a particular goal. While this flexibility can be restrained by standardizing your encoding according to a DTD or schema, there remains the potential for endless variation.

Because of this, web browsers are unable to do anything more than to check for well-formedness and display the hierarchical outline of your document. (These abilities should not be minimized!) They understand only one set of tags—HTML, so if you plan on using your XML documents in a web application, they will, at some point in the process, have to be converted to HTML. This is where XSLT comes in.

XSLT (Extensible Stylesheet Language for Transformation) is an XML-based technology that allows you to take a source document (as long as it is well-formed), and by applying a set of template-based rules, create, amongst other possibilities, a new XML or HTML file. While the particulars of the XSLT syntax are beyond the scope of this article, the basic processing model of an XSLT stylesheet is to look for a node in the source document (or tree), and then establishing what to output when that node is found. Using our CD document, for example, we would look for the <title> node within an <albuminfo> node, and
when we find one, output its contents between <h2> tags, and look for the <title> node within the <track> node and output it between <font color="blue"> tags. In so doing, we work with the structural knowledge embedded in the source document (referred to as the source tree) and apply presentational rules that permit browsers to display the selected data.

There are a variety of ways you can handle these sorts of transformations. You can send the entire XML document to the user’s browser with an embedded stylesheet reference. Recent versions of the major browsers have built-in transformation engines (such as msxml in the Internet Explorer browser), and these browsers can perform the transformations on the client workstation and display only the acceptable result of the transformation. Alternatively, you can process the XSLT instructions on the server (via a servlet or CGI call) and return not the source document, but the results of the transformation. Some prefer to use XML as their storage medium, and run the XSLT transformations as part of publishing their documents to the web site, so that the user never encounters pure XML, but only the results of an a-priori processing.

Putting this all together, we can visualize the overall XML environment (greatly simplified, of course) in this manner:
WORKS CITED