RC 21923 (98573) 28 December 2000 Computer Science

IBM Research Report

Supporting XML Views on Distributed and Heterogeneous Data Sources

Ming-Ling Lo, Shyh-Kwei Chen, Sriram Padmanabhan
IBM Research Division
Thomas J. Watson Research Center
P.O. Box 704
Yorktown Heights, NY 10598

Research Division
Almaden - Austin - Beijing - Haifa - T. J. Watson - Tokyo - Zurich

Supporting XML Views on Distributed and Heterogeneous Data Sources

Ming-Ling Lo, Shyh-Kwei Chen, Sriram Padmanabhan
IBM T.J. Watson Research Center
{mllo,skchen,srp}@us.ibm.com

Abstract

XML has emerged as an important data representation and exchange format. Increasingly new applications in areas such as e-business require access of data in XML format regardless of their origins in distributed and heterogeneous data sources. In this paper, we introduce a mapping framework for supporting virtual XML documents, or XML views, on top of distributed and heterogeneous data sources. The framework allows very flexible mapping from underlying data to XML, supports arbitrary underlying data models, and constructs virtual XML documents using only high level declarative descriptions. The user's intension in mapping is described by annotating DTDs. Each annotated DTD logically represents a set of virtual XML documents. The virtual XML documents can be retrieved from, deposited into, and queried against the set. Underneath we use a two-stage approach. The front-end stage assumes a data model of only lists and scalars, and focuses on enabling flexible mapping from this model to XML. The back-end stage implements the conceptually simpler task of mapping from underlying data sources to the intermediate lists and scalars, and encapsulates all the system and data model heterogeneity.

Keywords: XML, DTD, schema, meta data, schema mapping, e-business, web applications

1 Introduction

XML has emerged as an important data representation and exchange format[1]. Increasingly new applications in areas such as e-business require access to data in XML format, existing applications are modified or extended to accept XML as input or output, and business processes exchange XML documents in their logic flow. Given this trend, it is generally acknowledged that the capability to access the large amount of existing data in various non-XML data sources as XML data becomes very important. Such data sources may include relational, hierarchical, or object databases, spreadsheets or other smaller data stores, and the outputs of various applications. It is often necessary to combine data from multiple sources into individual XML documents. For example, an XML product report may combine sales and price information from a relational database in the sales department, inventory information from the supply-chain system, product

description from an object-oriented databases in the R&D department, etc.

In this paper, we introduce a framework, called the DTDSA framework, that supports XML views for distributed and heterogeneous data sources. Our framework is designed with flexibility and ease of use as two guiding principles. Flexibility reflects in two aspects. The mapping mechanism designed should not impose any restriction on the fashion in which a set of underlying data can be mapped into XML. This enables users to map data in ways that best suit their needs. In addition, there should also be no limitation on the types of data models or systems that can serve as underlying data sources. Ease of use reflects in three aspects. The mapping logic should be described in a very high-level, declarative syntax; the software embodying the framework should be very easy to deploy; and the supported XML views, or virtual XML documents, should incur little costs to create, manage and use. Such a project involves extensive efforts in both framework design and system implementations. This paper focuses on the former.

While there are many approaches to the view definition problem including custom, we believe that the general mediator approach is most useful. A mediator could use a query language method for describing views as in SilkRoute[2] or XPERANTO[3] or it could use an annotation/functional specification approach such as XDuce[4] or the DAD specification from IBM DB2 XML Extender[5]. We focus on the annotation approach in our work for the following reasons: (i) it is easier to adapt to heterogeneous data sources, (ii) it provides a rigorous interpretation of the elements that compose the larger XML view, and (iii) these element annotations can be reused in other or combined in multiple XML views using query or other mechanisms.

To map a wide variety of underlying data models and systems, we use an approach that consists of two conceptual stages. In the first stage, called the front-end mapping, we assume that all underlying data consists of only lists and scalars, and focus on mapping these values to XML. Most XML specific issues are handled in this stage. In the second stage, called the back-end mapping, individual data sources are mapped to lists and scalars. System and data model heterogeneity is encapsulated in this stage. Dividing the task in this way not only simplifies the overall system design complexity, but also avoids repeating efforts for each new type of data sources.

We base our mapping description on the Document Type Definition (DTD), which is part of the XML specification[1]. However, all discussions in this paper are equally applicable to the XML Schema[6] which serves similar purpose as DTD but is more sophisticated. Our approach follows the observation that a DTD is an equivalent description of an XML document, if certain additional information is available. Our framework allows one to systematically supplement that information to achieve a mapping by inserting only two types of simple mapping constructs into DTDs. Finally, for the same mapping specification to be applicable for XML retrieval, deposit and query purposes, the mapping must be declarative rather than procedural.

The rest of the paper is organized as follows. Section 2 explains the motivation, terminology, and notation of this work, and discusses related work. Section 3 introduces the DTDSA framework, including the functional model of DTD, and the syntax and semantics of the introduced mapping constructs. Section 4 explains how the underlying data sources are integrated into the framework. Section 5 explains how the tagged data

PO

POID	BUYER	SELLER
100	20	10

company

COID	NAME	ADDR
10	IBM	NY
20	CITIBANK	NY

lineitem

POID	PRODID	AMOUNT
100	35678	20k
100	35694	100k

product

PRODID	NAME
35678	THINKPAD
35694	SERVER

Figure 1: An example purchase order relational schema.

PD_DTD

```
<!ELEMENT PD (pname, pdesc)>
<!ELEMENT pname (#PCDATA)>
<!ELEMENT pdesc (#PCDATA)>
```

documents

```
<PD>
<pname> THINKPAD </pname>
<pdesc> very good </pdesc>
</PD>
```

```
<PD>
<pname> SERVER </pname>
<pdesc> the best </pdesc>
</PD>
```

Figure 2: An XML repository containing a product description DTD PD and two sample documents.

interface can be integrated into the framework. Section 6 concludes the paper.

2 Preliminary

2.1 Motivation

Consider a purchase order relational schema with four tables, PO, company, lineitem and product (see Figure 1), and an XML repository containing a product description DTD PD_DTD and two XML documents (see Figure 2). In many situations, it is desirable to access the information in these two data sources as an integral XML document. For example, in the following XML document, the transaction information comes from the relational database in Figure 1, while the product description comes from the XML repository in Figure 2.

```
<PO>
    <id>100</id>
    <buyer>
        <name>CITIBANK</name> <address>NY</address>
    </buyer>
    <seller>
        <name>IBM</name> <address>NY</address>
    </seller>
    <lineitem>
        <prodname>THINKPAD</prodname>
        <proddesc>very good</proddesc>
        <amount>20K</amount>
    </lineitem>
    <lineitem>
        <prodname>SERVER</prodname>
        <proddesc>the best</proddesc>
```

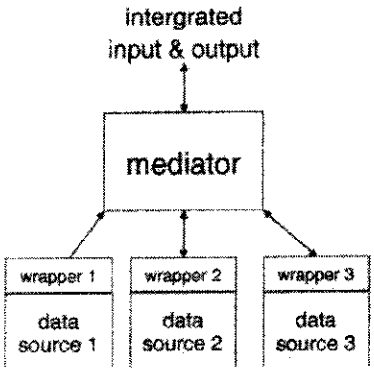

Figure 3: A generalized mediator-wrapper architecture.

```
        <amount>100K</amount>
    </lineitem>
</PO>
```

There are many different ways in which such an XML document can be created. In the most straightforward case, one can write a custom program to retrieve and assemble the data into the XML document. Such an approach lacks flexibility and is inefficient in terms of implementation effort. If we decide to present the document using a different DTD, add or remove a few elements from the document, or incorporating information from a new data source into the document, the program will need to be changed. Hence the challenge is in implementing a system in which different changes and requirements can be easily met.

Like many other systems, our solution uses the mediator[7] and wrapper architecture (see Figure 3.) Such an architecture encompasses a spectrum of solutions. At one extreme, in the *thick mediator-thin wrapper* approach, the mediator accesses the underlying data in their native formats and handles most of the heterogeneity and complexity of the underlying systems. The mediator in this approach may be quite complicated, while the wrappers serve little or no function. At the other extreme, in the *thin mediator-thick wrapper* approach, the mediator accesses all data sources through a uniform interface supported by the wrapper layer. The mediator in this case only needs to map high-level user requests into equally high-level local requests and collect the local result in a straightforward manner, while the wrapper must mask away all the data source complexity and heterogeneity.

Our solution uses a thin wrapper approach. However, we believe that with proper design, not only can both the mediator and the wrappers be light-weight, but we can also have great expressive power and flexibility of the system at the same time. One possibility to provide XML views on heterogeneous data sources is to use XML as the interface between the mediator and the wrappers, and uses XML transforming technologies such as XSLT[8] in the mediator to produce the target document. The advantage of such an approach is that many data sources have either added or are planning to add XML interfaces for accessing their data.

We do not use this approach for the following reasons. First, despite all the efforts, we do not expect all data sources of interest to us to have XML interfaces. In fact our system can serve as a universal XML interface for the data sources without XML interfaces. Second and more important, the nature of the problem is such that even if the underlying data sources provide XML interfaces, a lot of efforts in decomposing, rearranging, and assembling data into XML will need to be repeated in the mediator. The XML format may also not be the best format for mediator to perform extraction and rearrangement of data into final target document. All these lead to inefficiency. Although our framework does not rely on the underlying data sources to support XML interfaces, it is able to interact with such interface when one is available and its use makes sense, as illustrated in the po example above.

2.2 Terminology and Notation

In the definition of DTD, many supporting concepts are introduced in the XML specification[1]. Unfortunately many of these concepts are not explicitly defined and named in the specification. In order to have a accurate discussion, we summarize and assign names to those concepts that are relevant to our work (see Table 1.) Note that no new concepts beyond those already used in the XML specification are listed here [1].

When an XML document conforms to the XML grammar, it is considered *well-formed*. When it in addition conforms to a Document Type Definition (DTD), it is *valid* with respect to the DTD. A DTD describes and constrains the structure of its member documents.

A number of concepts are particularly important to our discussion. A *DTD construct* is a building block for DTD. They are the parts of the DTD text that can be uniquely identified. Using the terminology defined in Table 1, a DTD construct can be an element name, a content unit, a content particle, a value declaration or a DTD declaration. A *value declaration* declares that a piece of text will take its place in the actual XML documents. In other words, it is a place holder for document content in a DTD. Value declarations include `#PCDATA` declarations and attribute type declarations. A *choice declaration* declares that one of the listed alternatives will take its place in the actual XML documents. Choice lists, terminal choice lists and enumerate type declarations fall in this category. Another frequently used term in DTD description is the *repetition symbols*, which include the symbols '?' (zero or once), '*' (zero or more times), and '+' (at least once), and are used to indicate the multiple occurrences of the preceding DTD construct. The following examples illustrate some of the terms described in Table 1.

- <!ELEMENT A ((B,C?)* | D+ | E) >

  (B,C?) : cu, cp; D+ : cp; E : cp; (...) : content-spec: choice

  A DTD declaration which is an element type definition (*ed*) for element `A`. The *content-spec* of this element type definition is a choice list of 3 alternatives that are content particles (*cp*), `(B,C?)*`, `D+`,

[1] To avoid creating confusion, we use names that either appear in the production rules of or conform to the style of the XML specification whenever possible. This may result in some less intuitive names unfortunately.

Term	Abbr.	Description
repetition symbol		'?', '*', or '+'
element name	*ename*	an element name
choice list	*choice*	a list of *cp* enclosed by '(' and ')', and separated by '|', i.e. "$(cp\|cp\|\cdots\|cp)$"
terminal choice list	*t-choice*	a list of *#PCDATA* or *enames*, each appearing only once, enclosed by '(' and ')', and separated by '|', i.e. "$(\#PCDATA\|\ ename\|ename\|\cdots\|ename)$"
enumeration type declaration		a list of keywords enclosed by '(' and ')' and separated by '|'. Used in attribute type declaration
choice declaration	*cd*	a choice list, terminal choice list, or enumeration type declaration
sequence	*seq*	a list of content particle enclosed by '(' and ')' and separated by ',', i.e, "$(cp, cp, \cdots, cp)$"
content unit	*cu*	An *ename*, *choice*, *seq*, or *t-choice*
content particle	*cp*	a content unit followed optionally by a repetition symbol
content spec	*content-spec*	the part that matches "content-spec" in the DTD production rules, i.e., the part that follows the element name and proceeds '>' in an element type declaration
PCDATA declaration		#PCDATA
attribute type declaration	*at*	the attribute type in an attribute list declaration, e.g., CDATA, ID, IDREF
attribute definition		the part that includes an *ename*, an *at*, and a default declaration
value declaration		PCDATA declaration or attribute type declaration
element type declaration	*ed*	the part that includes a "<!ELEMENT", followed by an *ename*, *content-spec*, and a '>'
attribute list declaration	*al*	the part that includes a "<!ATTLIST", followed by an *ename*, a list of attribute definitions, and a '>'
DTD declaration		element type declaration and attribute list declaration
DTD construct		a DTD declaration, content spec, value declaration, *cu*, *cp*, or *ename*

Table 1: Definition of DTD constructs and DTD related terms. The column "Abbr." is the abbreviations of the names given for the concepts.

and E. The first content particle is a content unit (*cu*) (i.e., (B,C?)), followed by a repetition symbol '*', where (B,C?) in turn represents a sequence. The second content particle is a *cu* (i.e., *ename* D) followed by a '+', while the last is a simple *cu* (i.e., *ename* E). The terms are recursively defined. For instance, sequence (B,C?), which is a *cu* itself, includes 2 *cp*'s B and C?. Note that *ename* B is both a *cp* and a *cu*, depending on the context in which we discuss it.

- <!ELEMENT A (#PCDATA | B | C)* > (*t-choice*)

  An element type declaration for *A* whose *content-spec* is a terminal choice list (*t-choice*), rather than a choice list, followed by a '*'.

- <!ATTLIST A B CDATA #IMPLIED (*attribute definition*)
  C (X | Y | Z) #REQUIRED > (*at: enumeration type declaration*)

  A DTD declaration which is an attribute list declaration (*al*) with 2 attribute definitions. The first attribute definition includes an *ename* B, an *at* CDATA, and a default declaration #IMPLIED, and the second includes an *ename* C, an *at* which is an enumeration type declaration (X|Y|Z), and a default declaration #REQUIRED.

2.3 Related Work

The SilkRoute[2] and XPERANTO[3] projects study mechanisms to generate XML views on underlying object-relational sources. Both techniques use a query mechanism such as XML-QL for describing the XML view. For the back-end, SilkRoute uses a language called RXL to describe the relational schema while XPERANTO uses SQL. Our system differs from these projects since its scope includes heterogenous and non-relational data sources. The XDuce[4] language is another technique that can be used to transform XML documents using functions that process regular expressions. In that sense, XDuce is similar to the DTDSA approach described in our paper. TSIMMIS[9], Garlic[10], DISCO[11], and InfoMaster[12] are prior examples of mediator-based research projects. Many of these projects are adapting to use XML schemas and meta data. Our system differentiates from these systems since it is based on XML schema or DTD at its very core and hence can provide more efficient processing and management of the XML views.

There have been a number of proposals for XML query languages such as XML-QL[13], XQL[14], and Quilt[15]. Our framework is query language neutral and can integrate with any of these languages to query the virtual XML documents. Also, our framework can utilize one of these query languages to access XML repositories that support the XML query interface as the underlying data sources.

There are also many XML view efforts at the single source level. Most database vendors have currently implemented schemes for supporting storage and retrieval of XML documents in decomposed or monolithic fashion[5, 16]. Different annotation schemes such as DAD from IBM DB2 or templates from Microsoft SQL

Server are being used to decompose XML documents to underlying relational schema elements. We believe that one can easily transform the annotations from DAD or other schemes into the DTDSA annotation model in order to access data through our system.

3 DTDSA Framework

Before using the underlying data as XML, the users must express their intentions regarding how the underlying data should be mapped. We called this process *authoring* of the mapping logic. Authoring is accomplished through a simple declarative annotation process in our framework.

3.1 DTDSA Framework and Functional Model

Our framework uses a two-stage approach, similar to that used by compiler technologies, to overcome the heterogeneity of data model and systems. The front-end stage assumes a simple data model in which there are only lists and scalars, and implements functions that map this data model to XML documents. The back-end stage implements the functions that map from arbitrary data sources to lists and scalars and encapsulates all heterogeneity. The front-end involves more conceptual complexity, while the back-end involves more system level complexity.

The center of the mapping framework is the simple list and scalar data model. In this model, a list consists of scalar elements. A scalar is either a *simple scalar object* or an *opaque object*. A simple scalar object is one that can be straightforwardly cast to a string value, including basic types such as integers and floating point numbers. An opaque object is one whose internal structure and semantics do not concern the front-end of the framework. They are treated as atomic units and are passed from one back-end function to another.

It is possible to design different description schemes to express the same mapping logic. We choose an approach based on annotating DTD for the following reasons. First, users are already familiar with the DTD syntax. It is easy to understand, and exempts users from learning yet another set of syntax. Second, using DTD as the basis of the mapping syntax guarantees the output XML documents to be valid with respect to the base DTD. In addition, mapping authoring process is simplified since one only needs to annotate instead of starting from scratch.

The DTD annotation is inspired by the following intuition. Given a DTD and an XML document conforming to it, the information in the DTD covers that in the document, provided that when we traverse from the root element of the DTD, whenever we encounter the following DTD constructs, the corresponding information is available:

Number of occurrences: For a repetition symbol, the number of times the preceding DTD construct repeats.

Choice: For a choice declaration, the alternative taken.

Data content: For a value declaration, the text content taking its place.

If we can add annotations to a DTD to supplement this information, the annotated DTD can be considered as equivalent to an XML document. If, furthermore, the annotations are parameterized, the annotated DTD can be equivalent to a set of XML documents. In our framework, the annotations are applied to specific DTD constructs. A DTD together with all its annotations is called a *Document Type Definition with Source Annotations (DTDSA)*. A DTD construct in a DTDSA with its corresponding annotations, if any, is called an *annotated DTD construct* or *DTDSA construct*.

Formally, we consider a DTDSA as equivalent to a function that takes a set of input parameters and returns an XML document. Specifically, we consider each annotated DTD construct as a function that returns a string, recursively assembled from the returned string of its child constructs. The function corresponding to an annotated DTD construct `C` is denoted by $F_C()$.

Definition: The function corresponding to a DTDSA `D` is the same as the function corresponding to its root element `R`, i.e. $F_D() = F_R()$. □

The range of a function `f` is denoted `Range(f)`. In a sense, a DTDSA `D` can also be considered as presenting `Range(`$F_D()$`)`.

Suppose `C` is an annotated DTD construct. If `C` is a sequence, its return value is the concatenation of those of its child DTD constructs. If `C` is a choice declaration, its return value is that of one of its child alternatives. An annotation will determine the choice from the alternatives. If `C` is `H#`, where `H` is a DTDSA construct and `#` is a repetition symbol, its return value is the concatenation of the return values of multiple invocations of `H`. Again, an annotation will determine the number of invocations. Finally, if `C` is a value declaration, such as a `#PCDATA` or a `CDATA`, an invocation of `C` simply returns a string determined by an annotation. The following subsections discuss where and how annotations may appear.

3.2 Value Specifications

Two types of annotations or *mapping constructs* exist in the DTDSA framework. They are *value specifications*, which answer the "data content" and "choice" questions, and *binding specifications*, which answer the "number of occurrences" questions and provide parameter values to mapping constructs. A mapping construct always contains a mapping function and annotates the DTD construct immediate preceding it. In this framework a constant value is treated as a constant function.

A value specification is a ':' followed by a function that returns a simple scalar object, called a *value function*. A value specification annotates either a value or choice declaration, and each value or choice declaration requires exactly one annotating value specification. A value specification appears in DTDSA as follows:

VC :*sf*

where VC is a value or choice declaration, and *sf* a scalar-valued function, called the *value function*.

3.2.1 Supplying values to value declarations

If VC is a value declaration, the semantics of the combination is that in every document instance of the DTDSA, the value of every occurrence of VC is value of sf.

Consider the following three one-element DTDs with value specifications already assigned to their value declarations as follows: [2]:

```
DTDSA NAME: <!ELEMENT NAME (#PCDATA :"John")>
DTDSA SALARY: <!ELEMENT SALARY (#PCDATA :10000+x)>
DTDSA AGE <!ELEMENT AGE (#PCDATA :f(y))>
```

In this examples DTDSA NAME always corresponds to the same XML document:

```
<NAME>John</NAME>
```

DTDSA SALARY corresponds to an infinite number of XML documents, depending on the value of x. When given x=20000, it corresponds to the XML document

```
<SALARY>30000</SALARY>
```

DTDSA AGE represents a particularly interesting case. The contents of the document instances depend both on the value of y and the definition of the function f(y). One may increase the domain of f(y) to increase the set of document instances, or change the definition of f(y) to change the document instance set. This type of mechanism is useful when mapping underlying data sources to virtual XML documents. Details of how the values are assigned to parameters of value functions are discussed in Section 3.3.

3.2.2 Determining alternatives for choice declarations

When VC is a choice declaration, the semantics of the combination is that in every document instance of the DTDSA, the alternative taken in very occurrence of VC is given by fs.

Suppose VC = (C1|C2|...|Cn), with annotation fs: VS:

```
(C1|C2|...|Cn) :fs
```

There are two possibilities. If the value produced by fs is an integer i, with i between 1 and n, the alternative taken in place of VC is Ci. Alternatively, if the value produced by fs is a string Cj which matches one of the alternatives C1, C2, ..., Cn, the alternative taken in place of CD is the matched alternative. Otherwise, the alternative taken is undefined. In actual implementations, a user defined default alternative or error reporting string can be used.

Consider the following example:

[2] For simplicity, DTD headers are omitted in all future discussions

```
DTDSA JOB_DESCRIPTION:
<!ELEMENT JOB_DESCRIPTION (SALES|RESEARCH) :f(x)>
<!ELEMENT SALES (#PCDATA :"Increase sales volume")>
<!ELEMENT RESEARCH (#PCDATA :"Develop new technology")>
```

where `f(x)` has the definition:

$$f(x) = \begin{cases} \text{"SALES"}, \text{when } x = 1 \\ \text{"RESEARCH"}, \quad \text{otherwise} \end{cases}$$

The XML document corresponding to `JOB_DESCRIPTION` given `x=1` is:

```
<JOB_DESCRIPTION>
<SALES> Increase sales volume </SALES>
</JOB_DESCRIPTION>
```

3.3 Binding Specifications

Any data construct that is not a value or choice declaration can have one or more associated binding specifications. However, except for content particles with ending repetition symbols, binding specifications are not mandatory. The order of binding specifications matters when more than one of them are associated with a DTD construct. The syntax of a binding specification is a "::" symbol, followed by a variable, a ":=" symbol, and a list-valued function. It appears in DTDSA as follows:

$$\boxed{\text{DC} :: x_1 := vf_1 :: x_2 := vf_2 \ldots :: x_n := vf_n}$$

where DC is a DTD construct which is not a value or choice declaration, x_i a variable and vf_i a binding function, for $i = 1, \ldots, n$. x_i is called a *binding variable*, and vf_i a *binding function*.

A binding specification may serve two purposes. First, when immediately following a repetition symbol it determines the actual number of repetitions for that repetition symbol in the document instances. Second, it may supply values to parameters of other value or binding functions that have the same variable name. Of course, its own binding function may contain parameters which in turn obtain values from other binding specifications. This feature allows a set of mapping constructs to relate to one another and forms the basis for combining a diverse set of data into individual XML documents in numerous ways.

Given a DTDSA specification, there are parameters in some mapping constructs that do not *always* obtain their values from other binding specifications. These parameters are called the *input parameters* of the DTDSA, and are used to identify specific documents among the set of document instances.

3.3.1 Extended containment and variable binding

To understand how binding variables work to supply values to various DTD constructs in a DTDSA specification, we can envision a DTDSA specification as a set of subroutines that constitutes a program. In this analogy, each DTD construct corresponds to a subroutine that takes some number of (explicit and implicit)

parameters as input and produces a piece of XML text as output. The process of generating an XML document can thus be viewed as calling the root element subroutine, which in turn called its child subroutines recursively.

Calling relationship between subroutines can be rigorously defined using the syntactic information of a DTD specification. The *extended containment relationship* is defined as follows. Given any DTD construct `D1`, the DTD construct `D2` that is a sub-expression of `D1` is said to be contained by `D1`. If any DTD construct that contains `D2` must also be `D1` or contain `D1`, then `D1` is called the *parent* of `D2`. In addition, any element name that appears in a certain DTD construct is also considered a parent of the element type declaration with the same element name. An element type declaration is considered the parent of the attribute list declaration with the same element name. The child relationship is defined as the reverse of the parent relationship, and transitive closure of the parent relationship is the extended containment relationship.

Consider the following DTD:

```
<!ELEMENT A (B, C)>
<!ELEMENT B ...>
<!ELEMENT C ...>
```

Its extended containment relationship includes an *element type declaration* (*ed* as in Table 1), with an *ename* `A`. The *ed* for `A` has a child sequence construct `(B, C)` which includes two children *cp* constructs `B` and `C`. One of the children *cp*'s, say `B`, has a child *cu* construct, which in turn includes a child *ename* `B`. The *ename* construct `B` is the parent of the *ed* construct `<!ELEMENT B ...>`

Note that the extended containment relationship in a DTD may contain cycles. However, cycles does not affect the correctness of the way DTDSA works. The following DTD contains a cycle:

```
<!ELEMENT A (#PCDATA|A)*>
```

Also note that some DTD construct may have more than one parents. For example, in the following DTD the element type definition of `D` has two parents, one being the element name `D` in `<!ELEMENT B (D)>`, the other that in `<!ELEMENT C (D)>`:

```
<!ELEMENT A (B, C)>
<!ELEMENT B (D)>
<!ELEMENT C (D)>
<!ELEMENT D (#PCDATA)>
```

The fact that one element type definition may have multiple parents follows naturally from the definition of DTD, and contribute to the modeling power of DTD. Using the subroutine paradigm, this simply means that a subroutine can be called by different callers.

For a DTD construct with an associated binding specification, its binding variable can be considered a location variable of the corresponding subroutine which will be passed to the input parameters of the child subroutine. All its child DTD constructs have, among others, an implicit input parameter with the same name.

When a value or binding function has a parameters x, it gets its value from the set of implicit input variables of the associated DTD construct. This leaves the question regarding the root element. The root element get its implicit input parameters from the input parameters of the whole DTDSA.

As an example, consider the previous DTD, now converted into a DTDSA:

```
1: <!ELEMENT A (B, C) ::x:=i1 ::y:=i2>
2: <!ELEMENT B (D) ::y:=x+10>
3: <!ELEMENT C (D) ::x:=x+20>
4: <!ELEMENT D (#PCDATA :x+y)>
```

When the input parameters of this DTDSA are i1=1 and i2=2, the corresponding document instance is derived as follows. Initially at line 1, x and y have the values of 1 and 2, respectively. y is redefined to 11 at line 2, while x is redefined to 21 at line 3. The #PCDATA at line 4 is called twice. In the context of A-B-D, x gets the value of 1, and y gets the values of 11, and the value of #PCDATA is thus 12. In the context of A-C-D, x is redefined to 21 while y remains at 2, and the value of #PCDATA is 23. The whole corresponding XML document is thus:

```
<A>
<B><D>12</D></B>
<C><D>23</D></C>
</A>
```

3.3.2 Number of repetitions

The repetition symbol can be envisioned as a for loop that calls the annotated DTD construct multiple times. The exact number of repetitions is jointly determined by the binding specification and the repetition symbol itself.

Let DC denote a DTD construct, x a variable, and vf a list-valued function producing a list of k values {v1, v2, ... vk}. The DTD construct with an associated binding specification, (DC)* ::x:= vf, can be considered as equivalent to the sequence (DC ::x:=v1, DC ::x:=v2, DC ::x:=vk).

Formally, given a DTD construct with an associated binding specification (DC)# ::x:=vf, where # is some repetition symbol, the DTD construct is considered equivalent to one of the following, depending on which repetition symbol # is:

1. For #=*:

   If $k \geq 1$, (DC)* is equivalent to k consecutive copies of DC. If $k = 0$ (i.e. vf evaluates to an empty list), (DC)* is equivalent to an empty string.

2. For #=+:

   If $k \geq 1$, (DC)+ is equivalent to k consecutive copies of DC. If $k = 0$, (DC)+ is equivalent to one copy of DC with x given an undefined value, i.e., equivalent to (DC ::x:=undefined).

3. For #=?:

If $k \geq 1$, (DC)? is equivalent to one copy of DC, and all except the first copy produced by vf are ignored, i.e., equivalent to (DC ::x:=v1). If $k = 0$, (DC)? is equivalent to an empty string.

When DC repeats more than once, each instance of DC sees a different binding of x. When DC is constrained to appear one (or zero) time, and vf produces a list of more than one values, only the first one (or zero) value is used. All other values in this case are ignored. On the other hand, if DC is required to appear at least once, but vf produces 0 values, the value of binding variable x is undefined. In actual implementations a user or system defined default value can be used.

Note that in these discussions, the symbol ":=" denotes neither equality nor simple assignment. Rather, it binds the list of values produced by the binding function one after another to the binding variable. As seen in the above definition, the number of values in the list affects the number of times the DTD construct preceding the repetition symbol repeats in the document instances.

Consider the following DTDSA:

```
<!ELEMENT A (B, C) ::x:=i1 ::y:=i2>
<!ELEMENT B (#PCDATA :y)>
<!ELEMENT C (D)* ::z:=intseq(x)>
<!ELEMENT D (#PCDATA :z)>
```

where the function intseq(x) produces a sequence of integers from 1 up to x. The virtual XML document corresponding to the DTDSA with i1=3, i2=5 is

```
<A>
<B>5</B>
<C> <D>1</D> <D>2</D> <D>3</D> </C>
</A>
```

4 Mapping Underlying Data

We have discussed the framework in terms of simple functions. This section discusses how a set of underlying data can be mapped to a set of XML documents in the framework.

The underlying data sources participate in the framework by implementing the mapping functions. In particular, they determine the domains, ranges, and definitions of the functions. We illustrate this point in a relational schema. Assume in a relational employee table EMP (see Figure 4), a function EmpWithJob(x) returns all rows in EMP whose JOB column has value x, and EmpName(y) returns the value of the NAME column of an employee row y. The state of the EMP table determines the domains, ranges, and definitions of the two functions.

Consider the DTDSA DEPT:

```
<!ELEMENT DEPT (EMPNAME* ::r:=EmpWithJob(x) ) >
<!ELEMENT EMPNAME (#PCDATA :EmpName(r)) >
```

If we set x='research' to the DTDSA, F_{DEPT}(x) returns:

EMP

ID	NAME	JOB
100	John	research
200	Mary	research
300	Joe	sales
400	Kathy	sales

Figure 4: A simple employee table.

```
<DEPT>
    <EMPNAME>John</EMPNAME>
    <EMPNAME>Mary</EMPNAME>
</DEPT>
```

Note that if the state of the EMP table changes, the definitions of the two mapping functions change as well. The definition of a DTDSA consists of both its syntactical representation (i.e., the base DTD plus the annotations) and the definitions of the mapping functions. Once the definitions of the mapping functions change, the definition of a DTDSA changes even though its syntactic representation may remain the same. This characteristics is used to implement the XML document deposit operation in the DTDSA framework.

4.1 Retrieval and Deposit Operations

We have so far defined DTDSA in terms of declarative functions. Such definition is useful only if it can be applied to the XML document retrieval and deposit operations. Retrieving XML documents using a DTDSA is straightforward. It can be achieved by simply supplying values to all of the unbound variables or input parameters, and evaluating the function corresponding to the DTDSA from the root element. If the contents of the underlying data sources are not changed, and the same set of parameter values is provided, the retrieval operation should return exactly the same XML document.

The XML document deposit operation using a DTDSA is less intuitive. It is defined as modifying the definitions of the mapping functions so that both the original and the newly entered data can later be retrieved. Consider the EMP table in Figure 4. Assume that we perform a deposit operation using the DTDSA DEPT, where x='research' and the input document d is

```
<DEPT>
    <EMPNAME>John</EMPNAME>
    <EMPNAME>Mary</EMPNAME>
    <EMPNAME>Jane</EMPNAME>
</DEPT>
```

A new row (500, 'Jane', 'research') will be inserted into EMP as the result, and the definitions of the functions change. F_{DEPT}('research') would now return the new document, while F_{DEPT}('sale') will give the same XML document as the one obtained before applying the deposit operation.

4.2 Opaque Objects

In the DEPT DTDSA, binding variable r is bound to rows of the table EMP. The value of r is not a simple scalar object, since we do not define how it can be cast into a string. However, this does not prevent the function F_{DEPT} from being well defined. This is because the value of r is not really processed by the DTDSA framework, but is passed directly to function **EmpName()**, which knows how to make use of the value. From DTDSA framework's point of view, the value of r is opaque, hence the name opaque object. As can be seen from the example, the design of list-valued binding function enables one to bring a big segment of underlying data into the DTDSA framework at a time, and the use of opaque objects enables one to use the underlying data freely. It greatly enhances the power of the framework without adding complexity to it.

4.3 Defining Mapping Functions

Theoretically any type of data sources can serve as an underlying data source for our framework. All that is required is to map an underlying data into lists and scalars through mapping functions.

The definitions of the mapping functions differ for different types of data models and data systems. Even for one particular type of data model, there may be multiple ways for defining the mapping functions. For example, in a relational schema, one may define a binding function based on a table to return lists of rows, or one may define a binding function based on a row to return lists of simple scalar objects which are its column values. Instead of exhaustively prescribing the best way to implement mapping functions for each type of data models, the integrators of data sources are free to define any mapping function that best suits their requirements.

The mapping functions completely encapsulate the heterogeneity of the data sources. Once the mapping functions are defined, the DTDSA framework treats all mapping functions the same. Users are free to mix and match mapping functions, including those based on different data models and systems in a single DTDSA.

Another issue is how mapping functions can be efficiently defined and named. This can be accomplished through functions that take the names of the underlying schema elements as input parameters, in addition to other input parameters. The mapping function can be generated by specifying the schema element names at the mapping authoring time.

For example, Let `row(T,C,x)` be a function that returns a list of rows in table `T` whose `C` column value equals `x`. If `T1` is a table in a relational database, `C1` is some column in `T1`, then `row(T1, C1, x)` can be used as a binding functions. Similarly, we can define `field(C, r)` as a function that returns the value of the column `C` in row `r`, and use `field(C1, r)` as a value function.

In a more general sense, one may simply regard a table as a list of rows, and any relational operation that returns a table as a potential binding function. Therefore, any SQL statement[17] can be used as a binding function.

4.4 Examples

In the following examples for convenience we use `T.C(r)` to denote `field(C,r)` for a row `r` from table `T`. Consider the `EMP` table in Figure 4. Suppose we want to enclose the employee name in a tag that is the employee's job title. We may use the `EMPLOYEE` DTDSA as follows:

```
<!ELEMENT EMPLOYEE (EMP_ID,
    (RESEARCH|SALES):EMP.JOB(r))> ::r:=sql("select * from EMP where ID=$x")
<!ELEMENT EMP_ID (#PCDATA :EMP.ID(r))>
<!ELEMENT RESEARCH (#PCDATA :EMP.NAME(r))>
<!ELEMENT SALES (#PCDATA :EMP.NAME(r))>
```

The function `sql()` executes the string argument as a parameterized SQL statement, and returns a list of rows, where the parameter `x` has a prefix '$'. The alternative taken in the choice list `(RESEARCH | SALES)` is determined by the value of `EMP.JOB(r)`. For `x = 300`, $F_{EMPLOYEE}$`(x)` returns:

```
<EMPLOYEE>
    <EMP_ID>300</EMP_ID>
    <SALES>Joe</SALES>
</EMPLOYEE>
```

It is possible to use more complicated binding and value functions. For example, the binding function can be a parameterized SQL statement involving joins, and the value function can combine values from multiple columns (of multiple tables) to produce a value.

Consider again the purchase order relational schema in Figure 1, and the XML repository in Figure 2. We can now build XML views on these two data sources using the DTDSA framework. An example is a po DTDSA as follows:

```
1. <!ELEMENT po (id, buyer, seller,
2.           (lineitem)* ::w:=row(lineitem, poid, PO.poid(r)))>
3.                       ::r:=row(PO, poid, x)
4. <!ELEMENT id (#PCDATA :PO.POID(r))>
5. <!ELEMENT buyer (name, address)> ::s:=row(company, id, PO.buyer(r))
6. <!ELEMENT seller (name, address)> ::s:= row(company, id, PO.seller(r))
7. <!ELEMENT name (#PCDATA :company.name(s))>
8. <!ELEMENT address (#PCDATA :company.addr(s))>
9. <!ELEMENT lineitem (prodname,
10.          proddesc ::d:=doc( PD_DTD, PD.pname, product.name(v)),
11.          amount)> ::v:=row( product, prodid, lineitem.prodid(w))
12.<!ELEMENT prodname (#PCDATA :product.name(v))>
13.<!ELEMENT proddesc (#PCDATA :PD.pdesc(d))>
14.<!ELEMENT amount (#PCDATA :lineitem.amount(w))>
```

In this DTDSA, all function parameters except `x` in line 3 get their values from some binding variables. `x` is thus the input parameter of the DTDSA. A binding variable is visible to the invocation of its descendant functions. For example, in line 3, the binding variable `r` annotates the element po, and is essentially visible to all the descendant invocations, while the binding variable `w` in line 2 annotates `(lineitem)*`, and is visible only to invocations triggered through that DTDSA construct.

The binding specifications in lines 5 and 6 share the same binding variable name s but have different binding functions. Their descendant DTD constructs may see different bindings depending on the context of the invocations. The elements **name** and **address** in lines 7 and 8 will take their values from the binding specification in line 5 if they are invoked in the context of **buyer**, and will take the values from line 6 if they are invoked in the context of **seller**.

The binding specification in line 10 is the key that links the data in relational tables and the documents in the underlying product description XML repository together. The documents bound to d are then used to supply values to element **proddesc** in line 13. Given x = 100, F_{po}(x) returns exactly the same XML document as in Section 2.1.

4.5 Hierarchical Function Evaluation

This section provides a hierarchical method for evaluating DTDSA functions. When the evaluation applies to the root element with given input values, and then recursively applies to descendant elements, the execution trace of the evaluated DTD constructs can serve as the traversal sequence for both XML document retrieval and deposit operations.

We use C to denote a DTD construct, M a mapping construct, and B a binding specification whose binding function produces a list of k values. We also use concepts of *binding* and *environment*, defined as follows:

Definition: A *binding* is a pair of binding variable x and value v, and is expressed as $x = v$. An *environment* is a set of bindings with the constraint that every variable name appears at most once. □

For example, $x = 1$ and $y = 2$ are bindings. The set $\{x = 1, y = 2, z = 3\}$ is an environment, while the set $\{x = 1, x = 2\}$ is not.

An important function used in the discussion is $eval(f, Env)$, which denotes the value of evaluating a function f given the bindings in the environment Env. For example, $eval(x + 1, \{x = 1, y = 2\}) = 2$, and $eval(concatenate(\text{"XM"}, x), \{x = \text{"L"}\}) = \text{"XML"}$.

Environments have an associated operation '+', which is left associative. Given an environment Env and a binding $x = v$, if Env does not contain any binding with binding variable x, $Env + \{x = v\}$ adds $x = v$ into the set of Env. If the Env already contains a binding with binding variable x, say $x = u$, the operation replaces $x = u$ with $x = v$ in Env.

In the context of DTD constructs and mapping constructs, '+' simply means concatenation. For example, suppose that C_1 and C_2 are DTD constructs, and M is a mapping construct. $C_1 = C_2 + M$ means C_1 is equivalent to C_2 followed by a mapping construct M. Another important function is $bnd()$, which denotes a specific binding in the series of bindings produced by a binding specification given an environment.

Definition: Suppose B is the binding specification :: $x := vf$, where vf produces a list of values $v_1, v_2, \ldots, v_n$ in the environment Env. $bnd(B, k, Env)$ denotes the kth binding specified by B in Env.

That is, $bnd(B, k, Env) = (:: x := eval(v_k, Env))$. □

We provide a formal definition of DTDSA as follows. Given the previous definitions, every DTD construct C is considered as a function returning XML fragments when evaluated in an environment Env. We achieve this by defining $eval(C, Env)$ in terms of lower level $eval()$ and $bnd()$ calls.

Given a DTDSA D with root element type declaration C_r, a set of bindings to the input parameters $\{p_1 = v_1, p_2 = v_2, \ldots, p_m = v_m\}$, and an environment Env containing this set of bindings, we simply define $D(v_1, v_2, \ldots, v_m) = eval(C_r, Env)$. The set of all XML document instances of the DTDSA is $\{eval(C_r, Env) | \forall Env\}$.

The key to a formal definition of DTDSA is thus the definition of $eval(C, Env)$ for all DTD construct C. They are defined recursively as follows:

Binding specification reduction: Let C_1 denote a DTD construct with associated binding specifications, and C_2 the same DTD construct without the binding specifications, i.e., $C_1 = C_2 + B_1 + B_2 + \ldots B_n$. If C_1 ends with a repetition symbol, then

$$eval(C_1, Env) = eval(C_2 + B_1, Env + bnd(B_n, 1, Env) + bnd(B_{n-1}, 1, Env) + \cdots + bnd(B_2, 1, Env)).$$

Otherwise,

$$eval(C_1, Env) = eval(C_2, Env + bnd(B_n, 1, Env) + bnd(B_{n-1}, 1, Env) + \cdots + bnd(B_1, 1, Env)).$$

ED: Let ed denote an element type declaration, T its tag name, *content* its content specification, and A_1, A_2, ...,A_i the annotated attribute list declarations associated with ed, (i.e., having the same element name as ed),

$$eval(ed, Env) = \begin{cases} \text{“}<T\text{”} + eval(A_1, Env) + eval(A_2, Env) & \\ \quad + \cdots + eval(A_i, Env) + \text{“}/>\text{”}, & \text{if content = “EMPTY”} \\ \text{“}<T\text{”} + eval(A_1, Env) + eval(A_2, Env) & \\ \quad + \cdots + eval(A_i, Env) + \text{“}>\text{”} & \\ \quad + eval(content, Env) + \text{“}</T>\text{”}, & \text{otherwise} \end{cases}$$

Note that a content specification by definition is also a content particle.

CP: Let cp be a content particle consisting of an annotated content unit cu (which can be a sequence, choice list, terminal choice list or an element name) followed by a repetition symbol.

1. If $cp = cu + \text{“}*\text{”}$,

$$eval(cp{+}B, Env) = \begin{cases} eval(cu, Env + bnd(B, 1, Env)) + eval(cu, Env + bnd(B, 2, Env)) & \\ \quad + \cdots + eval(cu, Env + bnd(B, k, Env)), & \text{for } k > 0 \\ \text{“”}, & \text{for } k = 0 \end{cases}$$

2. If $cp = cu + \text{“+”}$,

$$eval(cp{+}B, Env) = \begin{cases} eval(cu, Env + bnd(B, 1, Env)) + eval(cu, Env + bnd(B, 2, Env)) \\ \qquad + \cdots + eval(cu, Env + bnd(B, k, Env)), & \text{if } k > 0 \\ undefined, & \text{if } k = 0 \end{cases}$$

3. If $cp = cu + \text{“?”}$,

$$eval(cp + B, Env) = \begin{cases} eval(cu, Env + bnd(B, 1, Env)), & \text{if } k \geq 1 \\ \text{“”}, & \text{otherwise} \end{cases}$$

CU: Let cu denote an annotated content unit. We have $eval(cu, Env) = eval(C, Env)$ where C is SEQ, CD, or ENAME with the same annotations as cu.

SEQ: Let seq denote a sequence and $cp_1, cp_2, \ldots, cp_k$ denote annotated content particles. If $seq = (cp_1, cp_2, \cdots, cp_k)$, then

$$eval(seq, Env) = eval(cp_1, Env) + eval(cp_2, Env) + \cdots + eval(cp_k, Env)$$

CD: Let cd denote a choice declaration that is choice list or terminal choice list, $C_1, C_2, \ldots, C_n$ the alternatives of cd, and V the value specification associated with cd, with $eval(V, Env) = v$.

$$eval(cd + V, Env) = \begin{cases} eval(v, Env), & \text{if } v \text{ is the same string as an alternative of } cd, \\ eval(C_v, Env), & \text{if } v = k, \text{ where } k \text{ is an integer and } 1 \leq k \leq n, \\ \text{undefined}, & \text{otherwise} \end{cases}$$

PCDATA: If the construct is #PCDATA + V, where V is a value specification
$eval(\#\text{PCDATA} + V, Env) = eval(V, Env)$

ENAME: Let C_1 be an element name that appears in some content specification, and ed be the corresponding annotated element type definition of C_1,

$$eval(C_1, Env) = eval(ed, Env)$$

AL: Let al be an attribute list declaration, and an_i and at_i be its ith attribute name and annotated attribute type declaration, respectively.

$$eval(al, Env) = \begin{cases} an_1 + \text{“=”} + eval(at_1, Env) + S+ \\ an_2 + \text{“=”} + eval(at_2, Env) + S + \cdots + \\ an_k + \text{“=”} + eval(at_k, Env) + S \end{cases}$$

where S represents white space.

ATD: Suppose at is an attribute type declaration, and V is the associated value specification with $eval(V, Env) = v$. If at is an enumeration type with n alternatives, i.e., $(a_1|a_2|\ldots|a_n)$:

$$eval(at + V, Env) = \begin{cases} v, & \text{if } v \text{ is the same string as an alternative} \\ a_k, & \text{if } v = k, \text{ where k is an integer and } 1 \leq k \leq n \\ \text{undefined}, & \text{otherwise.} \end{cases}$$

If at is not an enumeration type:

$$eval(at + V, Env) = v$$

5 Mixing XML Data and Non-Tagged Data

Regardless of whether an underlying data source is an XML repository or a non-XML data source such as a relational database system, it is possible to extract both XML and non-XML data from the data source. For example, a relational database may store a whole block of XML text as a large object (LOB) in a single column, which may be retrieved as a whole. On the other hand, we may retrieve the text content without the tags from a leaf level element of an XML document, as we did in Section 4.4. Regardless of the type of data sources accessed and the type data stored in it, we call the mechanisms that access a data source in the form of tagged XML data the *tagged data interface*, and those that access a data source in the form of non-tagged data the *non-tagged data interface*.

So far we have used the non-tagged data interface exclusively. There may be situations in which one wants to map segments of underlying XML data directly into various parts of the target XML document. The tagged data interface can be incorporated into DTDSA to satisfy this need.

The DTDSA framework can be modified to accommodate mixed interfaces. For DTDSA with mixed interfaces, a value specification may be associated with any DTD construct. Value specifications of non-tagged data interface must still be associated with data declarations. Value specifications with tagged data interface are associated with DTD constructs which are not data declarations. For the latter, the XML text segment returned by a value function becomes the result of evaluating the associated DTD construct, provided that the XML segment conforms to the structural constraints imposed by the DTD construct and its descendants. Otherwise, the result of evaluating this DTD construct is undefined.

XML-valued value specification: Let C be a DTD construct that is not a data declaration, and V be the associated value specification with $eval(V, Env) = v$, where v is an XML text segment.

$$eval(C + V, Env) = \begin{cases} v, & \text{if } v \text{ satisfies the XML structural constraints.} \\ \text{undefined}, & \text{otherwise.} \end{cases}$$

6 Conclusions

We have presented the DTDSA framework to generate flexible XML views conforming to the application or user's view of the data from heterogeneous data sources. Our method follows the observation that the information contained in a DTD covers that contained in an XML document if certain additional information is available. We introduce two types of mapping constructs, binding specifications and value specifications, to describe this additional information. The mapping constructs can contain parameters that are interrelated, and can be used to express mappings flexibly. The mapping constructs rely only on the availability of scalar-valued and list-valued functions. This simplicity enables applying our mapping scheme to a large variety of underlying data sources. Since the DTDSA framework is declarative, the same DTDSA can be used for XML document retrieval, deposit, and query. We have demonstrated the feasibility of the DTDSA framework by applying it to underlying relational databases and XML repositories, and illustrated how various mappings to such data sources can be composed.

As a proof of concept, we have implemented a prototype of the DTDSA framework using Java that retrieves and deposits XML documents. Currently accepted underlying data sources include any JDBC[18] compliant data sources, Lotus Notes databases, the IBM IMS hierarchical database system, and the XML repositories that support XPath[19]. Our implementation can access new data sources seamlessly, with a simple interface that allows the users to define their own mapping functions and to plug-in their actual implementations of the functions. A snapshot of the prototype has been posted on the IBM alphaWorks Web site, under the name *XML Lightweight Extractor (XLE)*[20], and has been widely tested and used.

In the future, we plan to investigate further issues involved with XML document deposit and query under the DTDSA framework. We are also interested in implementing DTDSA in a large-scaled system, conducted in the context of the XAS project[21]. As DTDSA scales, there will also be new issues such as managing and growing the internal intelligence about mapping and using fragments of existing DTDSAs as building blocks of new XML views.

References

[1] T. Bray et al., "Extensible Markup Language (XML) 1.0" W3C Recommendation. `http://www.w3.org/TR`.

[2] M. Fernandez, W.-C. Tan, and D. Suciu, "SilkRoute: Trading between Relations and XML," in *Proc. of WWW9, 9th Intl. World Wide Web Conference*, 2000. `http://www9.org/w9cdrom`.

[3] M. Carey et al., "XPERANTO: Publishing Object-Relational data as XML," in *Proc. of the 3rd Intl. Workshop on Web and Databases, WebDB 2000*, pp. 105–110, 2000.

[4] H. Hosoya and B. Pierce, "XDuce: A typed XML processing language," in *Proc. of the 3rd Intl. Workshop on Web and Databases, WebDB 2000*, pp. 111–116, 2000.

[5] IBM Corp., "DeveloperWorks: XML zone." http://www.ibm.com/developer/xml.

[6] H. S. Thompson et al., "XML Schema." W3C working draft, April 2000. http://www.w3.org/XML/Schema.

[7] G. Wiederhold, "Mediators in the architecture of future information systems," *IEEE Computer*, pp. 38–49, March 1992.

[8] J. Clark, "XSL transformations (XSLT) specification 1.0" W3C Recommendation. http://www.w3.org/TR/xslt.

[9] Y. Papakonstantinou, H. Garcia-Molina, and J. Widom, "Object exchange across heterogeneous information sources," in *Proc. International Conference on Data Engineering*, pp. 251–260, 1995.

[10] M. Carey et al., "Towards heterogeneous multimedia information systems: the Garlic approach," in *Proc. the Fifth International Workshop on Research Issues in Data Engineering (RIDE): Distributed Object Management*, 1995.

[11] A. Tomasic, L. Raschid, and P. Valduriez, "Scaling Heterogeneous Databases and the design of Disco," in *Proc. of Intl. Conf. on Distributed Computing Systems*, pp. 449–457, 1996.

[12] M. Genesereth, A. Keller, and O. Duschka, "InfoMaster: An information integration system," in *Proc. of ACM SIGMOD Conference*, pp. 539–542, 1997.

[13] A. Deutsch et al., "XML-QL: A query language for XML," 1998. http://www.w3.org/TR/NOTE-xml-ql/.

[14] J. Robie, J. Lapp, and D. Schach, "XML query language (XQL)," 1998. http://www.w3.org/TandS/QL/QL98/pp/xql.html.

[15] D. Chamberlin, J. Robie, and D. Florescu, "Quilt: An XML query language for heterogeneous data sources," in *Proc. of the 3rd Intl. Workshop on Web and Databases, WebDB 2000*, pp. 53–62, 2000.

[16] Oracle Corp., "Oracle8i." http://www.oracle.com/database/oracle8i.

[17] J. Melton and A. R. Simon, *The new SQL: a complete guide*. Morgan Kaufmann, 1993.

[18] Sun Microsystems, "JDBC Data Access API." http://www.javasoft.com/products/jdbc/.

[19] J. Clark, "XML path language (XPath)" W3C Recommendation. http://www.w3.org/TR/xpath.

[20] IBM Corp., "XML lightweight extractor (XLE)." http://www.alphaworks.ibm.com/tech/xle.

[21] M.-L. Lo, S.-K. Chen, S. Padmanabhan, and J.-Y. Chung, "XAS: a system for accessing componentized, virtual XML documents," tech. rep., IBM T. J. Watson Research Center, 2000.