Running head: USING LOGIC

# Using Logic for Querying XML Data

Nick Bassiliades, Ioannis Vlahavas
Dept. of Informatics
Aristotle University of Thessaloniki
54006 Thessaloniki, Greece
{nbassili,vlahavas}@csd.auth.gr

Dimitrios Sampson
Informatics and Telematics Institute
1 Kyvernidou Str.
54639 Thessaloniki, Greece
sampson@ath.forthnet.gr

## Abstract

In this chapter, we propose the use of first-order logic, in the form of deductive database rules, as a query language for XML data and we present X-DEVICE, an extension of the deductive object-oriented database system DEVICE for storing and querying XML data. XML documents are stored into the OODB by automatically mapping the DTD to an object schema. XML elements are treated either as classes or attributes based on their complexity, without loosing the relative order of elements in the original document. Furthermore, this chapter describes the extension of the system's deductive rule query language with second-order variables, general path and ordering expressions, for querying over the stored, tree-structured XML data and constructing XML documents as a result. The extensions were implemented by translating all the extended features into the basic, first-order deductive rule language of DEVICE using meta-data about stored XML objects.

# Using Logic for Querying XML Data

## Introduction

The success of the Internet depends on the availability of applications that will offer valuable e-services. However, applications have always been depended on input data and most importantly on their well-structuredness. So far, information is captured and exchanged over Internet through HTML pages, without any conceptual structure. XML is the currently proposed standard for structured or even semi-structured information exchange over the Internet (W3 Consortium, Oct 2000). However, the maintenance of this information is equally important. Integrating, sharing, re-using and evolving information captured from XML documents are essential for building long-lasting applications of industrial strength.

The story of information management or data management has been told before in the form of DBMSs. Over than three decades of research have been devoted to developing theory and systems for capturing, storing, maintaining and retrieving data for a single or multiple users. Such a vast research and development wealth should be re-used with the minimum of effort for managing semi-structured data, i.e. XML or SGML, which is the super-set of XML. There already exist several proposals on methodologies for storing, retrieving and managing semi-structured data stored in relational, object-relational and object databases. Furthermore, there exist quite a few approaches in storing SGML multimedia documents in object databases.

Capturing XML data in traditional DBMSs is one aspect of the story. Effective and efficient querying and publishing these data on the Web is another aspect that is actually more important since it determines the impact this approach will have on future Web applications. There have been several query language proposals (Abiteboul, Quass, McHugh, Widom, & Wiener, 1997; Buneman, Fernandez, & Suciu, 2000; Chamberlin, Robie, & Florescu, 2000; Deutsch, Fernandez, Florescu, Levy, & Suciu, 1999; Hosoya & Pierce, 2000; Robie, Lapp, & Schach; W3 Consortium, Dec 2001d) for XML data. Furthermore, recently the WWW consortium issued a working draft proposing XQuery (W3 Consortium, Dec 2001a), an amalgamation of the ideas present in most of the proposed XML query languages of the literature. Most of them have functional nature and use path-based syntax. Some of them (Abiteboul et al, 1997b; Chamberlin et al., 2000; Deutsch et al., 1999a), including XQuery, have also borrowed an SQL-like declarative syntax, which is popular among users. Some of the problems relating to most of the above approaches is the lack of a comprehensible data model, a simple query algebra, with the exception of (Buneman et al., 2000; Hosoya & Pierce, 2000) and query optimization techniques. There are proposals for a data model (W3 Consortium, Dec 2001b) and a query algebra (W3 Consortium, Jun 2001) for XQuery, however it is not yet clear how these will lead to efficient data storage and query optimization.

In this chapter, we propose the use of deductive rules as a query language for XML data and we present X-DEVICE, a deductive object-oriented database for managing XML data. X-DEVICE is an extension of the active object-oriented knowledge base system DEVICE (Bassiliades, Vlahavas, & Elmagarmid, 2000). DEVICE integrates high-level, declarative rules (namely deductive and production rules) into an active OODB that supports only event-driven rules (Diaz & Jaime, 1997), built on top of Prolog. This is achieved by translating each high-level rule into one event-driven rule. The condition of the declarative rule compiles down to a set of complex events that is used as a discrimination network that incrementally matches the rule conditions against the database.

X-DEVICE extends DEVICE by incorporating XML data into the OODB by automatically mapping DTDs of XML documents to object schemata, without loosing the document's original order of elements. XML elements are represented either as first-class objects or as attributes based on their complexity. Furthermore, X-DEVICE extends the deductive rule language of DEVICE with new operators that are used for specifying complex queries and materialized views over the stored semi-structured data. Most of the new operators have a second-order syntax (i.e. variables range over class and attribute names), but they are implemented by translating them into first-order DEVICE rules (i.e. variables can range over class instances and attribute values), so that they can be efficiently executed against the underlying deductive object-oriented database.

The advantages of using a logic-based query language for XML data come from the well-understood mathematical properties and the declarative character of such languages, which both allow the use of advanced optimization techniques, such as magic-sets. Furthermore, X-DEVICE compared to other XML functional query languages (e.g. XQuery) has a more high-level, declarative syntax that allows users to express everything that XQuery can express, in a more compact and comprehensible way, with the powerful addition of general path expressions, which is due to fixpoint recursion and second-order variables. Using X-DEVICE, users can express complex XML document views, a fact that can greatly facilitate customizing information for e-commerce and/or e-learning. Furthermore, the X-DEVICE system offers an inference engine that supports multiple knowledge representation formalisms (deductive, production, active rules, as well as structured objects), which can play an important role as an infrastructure for the impending semantic Web (W3 Consortium, Nov 2001).

In this chapter, we initially overview some of the related work done in the area of storing and querying XML data in databases, and then we describe the mapping of XML data onto the object data model of X-DEVICE. Based on this model, we present the X-DEVICE deductive rule language for querying XML data through several examples that have been adopted from the XML Query working group (W3 Consortium, Dec 2001c). Finally, we conclude this chapter and discuss future work.

# Related Work

*Managing XML data*

There exist two major approaches to manage and query XML documents. The first approach uses special purpose query engines and repositories for semi-structured data (Buneman, Davidson, Hillebrand, & Suciu, 1996; Goldman & Widom, 1997; Lucie Xyleme, 2001; McHugh, Abiteboul, Goldman, Quass, & Widom, 1997; Naughton et al., 2001). These database systems are built from scratch for the specific purpose of storing and querying XML documents. This approach, however, has two potential disadvantages. Firstly, native XML database systems do not harness the sophisticated storage and query capability already provided by existing database systems. Secondly, native XML database systems do not allow users to query seamlessly across XML documents and other (structured) data stored in database systems.

Traditional data management has a vast research background that cannot be just thrown away. The second approach to XML data management is to capture and manage XML data within the data models of either relational (Deutsch, Fernandez, & Suciu, 1999; Florescu & Kossmann, 1999; Schmidt, Kersten, Windhouwer, & Waas, 2000; Shanmugasundaram et al., 1999), object-relational (Hosoya & Pierce, 2000; Shimura, Yoshikawa, & Uemura, 1999) or object databases (Chung, Park, Han, & Kim, 2001; Nishioka & Onizuka, 2001; Renner, 2001; Yeh, 2000). Our system, X-DEVICE, stores XML data into the object database ADAM (Gray, Kulkarni, & Paton, 1992), because XML documents have by nature a hierarchical structure that better fits the object model. Also references between or within documents play an important role and are a perfect match for the notion of object in the object model. This better matching between the object and document models can also be seen in the amount of earlier approaches in storing SGML multimedia documents in object databases (Abiteboul et al., 1997a; Abiteboul et al, 1997b; Ozsu et al., 1997).

Relational mapping approaches

When XML data are mapped onto relations there are two major limitations: First, the relational model does not support set-valued attributes, therefore when an element has a sub-element with the multiple-occurrence expressions star (*) or cross (+), the sub-element is made into a separate relation and the relationship between the element and the sub-element is represented by introducing a foreign key. The querying and reconstruction of the XML document requires the use of "expensive" SQL joins between the element and sub-element relations. On the other hand, object databases support list attributes; therefore, sub-elements (or rather references to sub-elements) can be stored with the parent element and retrieved in a non-expensive way.

A second limitation of relational databases is that in order to represent relationships between elements-relations, join attributes should be created manually. On the other hand, in object databases,

relationships between element-classes are represented in the schema by object-referencing attributes (pointers), which are followed for answering queries with path expressions.

Finally, another problem is that relations are sets with no ordering among either their attributes or tuples. However, in XML documents ordering of elements is important, especially when they contain textual information (e.g. books, articles, Web page contents). Of course, there exist some relational approaches that hold extra ordering information in order to be able to reconstruct the original XML documents. However, these approaches add extra complication to the relation schema, query processing and XML reconstruction algorithms.

Object-oriented mapping approaches

Object database and some of the object-relational approaches to storing and querying XML data usually treat element types as classes and elements as objects. Attributes of elements are treated as text attributes, while the relationships between elements and their children are treated as object referencing attributes. There are some variations of the above schema between the various approaches. For example, in (Abiteboul et al., 1997a) and (Yeh, 2000) all the elements are treated as objects, even if their content is just PCDATA, i.e. mere strings. However, such a mapping requires a lot of classes and objects, which wastes space and degrades performance, because queries have to traverse more objects than actually needed. In (Nishioka & Onizuka, 2001) and in X-DEVICE this problem is avoided by mapping PCDATA elements to text attributes.

Moreover, some other approaches (Boehm, Aberer, Neuhold, & Yang, 1997; Hosoya & Pierce, 2000; Ozsu et al., 1997) go further by treating some elements with an internal structure as text attributes. The internal structure of these elements can only be accessed through special XML-aware text processing methods. The decision on which elements should be treated as classes or text attributes is either left on the database designer (Boehm et al., 1997; Ozsu et al., 1997) or it is heuristically taken based on data usage and query statistics (Hosoya & Pierce, 2000). This approach can sometimes prove more efficient regarding storage space requirements and faster query processing due to less fragmentation of elements, however the querying process is more complex because different access methods may be used for different portions of the same path expressions. Furthermore, the implementation requires the extension of the object database itself to handle such XML-aware attributes and possibly the extension of the basic object querying language to be aware of such attributes.

On the other hand, the mapping scheme we employed in X-DEVICE does not require any extension of either the database primitives or the basic query language. X-DEVICE is smoothly integrated with the existing DEVICE system by translating every new construct into one or more rules of the basic deductive rule language.

The in-lining approach that has been proposed in (Shanmugasundaram et al., 1999) for relational databases is followed in (Chung et al, 2001) to avoid producing too many classes in the

schema. However, the rationale for the in-lining method in (Shanmugasundaram et al., 1999) was that it reduces the amount of tables and joins between them, while in object databases there are no joins and therefore there is no rationale for using it. Furthermore, the resolution of path expressions gets complicated since some parts of the path consist of simple attribute names, while some others consist of path fragments that are "in-lined" as a single attribute in a great master class. The decision on which parts are simple or complex is based on five rules.

Handling of alternation

Another major issue that must be addressed by any mapping scheme is the handling of the flexible and irregular schema of XML documents that includes alternation elements. Some mapping schemes, such as (Nishioka & Onizuka, 2001) and (Shanmugasundaram et al., 1999), avoid handling alternation by using some simplification rules, which transform alternation to sequence of optional elements: `(X|Y)->(X?,Y?)`. However, this transformation, along with some other ones, does not preserve equivalence between the original and the simplified document. In the previous simplification rule, for example, the element declaration on the left-hand side accepts either an `X` or an `Y` element, while the right-hand side element declaration allows also a sequence of both elements or the absence of both.

Alternation is handled by union types in (Abiteboul et al., 1997a), which required extensions to the core object database $O_2$. This approach is efficient, however it is not compatible with the ODMG standard (Cattell, 1994) and cannot easily be applied in other object database systems. In X-DEVICE instead of implementing a union type, we have emulated it using a special type of system-generated class that its behavior does not allow more than one of its attributes to have a value. Furthermore, the parent-element class hosts aliases for this system generated class, so that path resolution is facilitated.

A similar approach has been followed in (Yeh, 2000). However, their typing system involves unnecessarily many class types. For example, multiple occurrences of elements are handled both by lists and a special class, which is different for a single-occurrence element. Furthermore, they handle even PCDATA elements as objects, which has certain drawbacks, as discussed above. Moreover, their effort does not include a special query language for the stored XML data, but only a visual query interface.

Finally, the representation of alternative and optional elements in (Chung et al, 2001) are uniformly handled as an inheritance problem. Specifically, there is one superclass for the element that contains optional and alternative elements and as many subclasses as the number of combinations of the alternative and optional elements. The superclass is an abstract class, i.e. it just "hosts" the mandatory sub-elements of an element, while the subclasses inherit the mandatory structure and define additional (optional and alternative) sub-elements. The basic advantage of using such a mapping scheme is that: a) null values are avoided since subclasses do not have optional attributes for optional

elements, and b) query processing is easily optimized by targeting only the subclass that satisfies the structure implied by the query.

One major problem with the inheritance approach to alternation of (Chung et al, 2001) is that it preserves ordering of elements only under simple types of alternation, while it is unclear what happens when the combination of optionality and alternation is considered for the same group of elements. For example, the method can handle a declaration like this: `a->(b,c*,(d|e))`, while it is unable to handle a declaration like the following: `a->(b,c,(d|e)*)`, without loosing the relative order of `d` and `e` elements in an XML document.

*Querying XML documents using logic-based languages*

Logic has been used for querying semi-structured documents, in WebLog (Lakshmanan, Sadri, & Subramanian, 1996) for HTML documents and in F-Logic/FLORID (Ludäscher, Himmeröder, Lausen, May, & Schlepphorst, 1998) and XPathLog/LoPix (May, 2001) for XML data. All these languages have a syntax that is influenced by F-Logic, a graph-based data model and are evaluated by bottom-up evaluation, similarly to X-DEVICE. None of them, however, supports generalized path expressions, alternation of elements and negation, features that are offered by X-DEVICE. Furthermore, X-DEVICE offers, in addition, incremental maintenance of materialized views when XML base data get updated.

WebLog

WebLog is a deductive Web querying language, similar in spirit to DATALOG, operating in an integrated graph-based framework. Although its syntax resembles F-Logic, it is not fully object-oriented. Navigation along hyperlinks is provided by built-in predicates, but navigation in the form of path expressions is not supported. Furthermore, there is no notion of generalized path expressions. Instead, recursive DATALOG-like rules replace regular expressions over attributes, attribute variables, and path variables. X-DEVICE offers both regular path expressions and generalized path expressions, i.e. path expressions with unknown number and names of intermediate steps.

F-Logic/XPathLog

F-Logic and its successor XPathLog are logic-based languages for querying semi-structured data. Their semantics are defined by bottom-up evaluation, similarly to X-DEVICE, however negation has not been implemented. Both languages can express multiple views on XML data; XPathLog, in addition, can export the view in the form of an XML document, much like X-DEVICE. However, none of the languages offers incremental maintenance of materialized views when XML base data get updated, as X-DEVICE does.

Both F-Logic and XPathLog are based on a graph data model, which can be considered as a schema-less object-oriented (or rather frame-based) data model. However, alternation of elements is not supported. Furthermore, F-Logic does not preserve the order of XML elements.

Both languages support path expressions and variables; XPathLog is based on the XPath language syntax (W3 Consortium, Dec 2001d). The main advantage of both languages is that, similar to X-DEVICE, they can have variables in the place of element and/or attribute names, allowing the user to query without an exact knowledge of the underlying schema. However, none of the languages supports generalized path expressions that X-DEVICE does, which compromises their usefulness as semi-structured query languages.

A noticeable feature of both languages is that they resolve IDREF attributes, linking them with the OIDs of object-elements that contain the equivalent ID attribute values. The implementation of such a feature requires strict typing in the document schema, a feature that DTDs currently lack, but the forthcoming XML Schema (W3 Consortium, May 2001) will support. The implementation of this feature with only DTD structuring information requires extra algorithms that are outside the scope of the basic mapping between the XML and object-oriented models. Nevertheless, this feature can be easily implemented in X-DEVICE.

Finally, XPathLog allows the declarative definition of updates in XML data, a feature that comes for free in X-DEVICE since the basic rule language supports multiple rule types (Bassiliades et al., 2000), such as deductive, production, event-based rules and rules for derived and aggregate attributes. Specifically, since XML data are smoothly captured in the object-oriented data model, database operations for objects immediately apply to XML data, as well. Furthermore, new operations specific to the tree-structured nature of XML can be supported. However, updating of XML documents will not be discussed further, since it is out of the scope of this chapter.

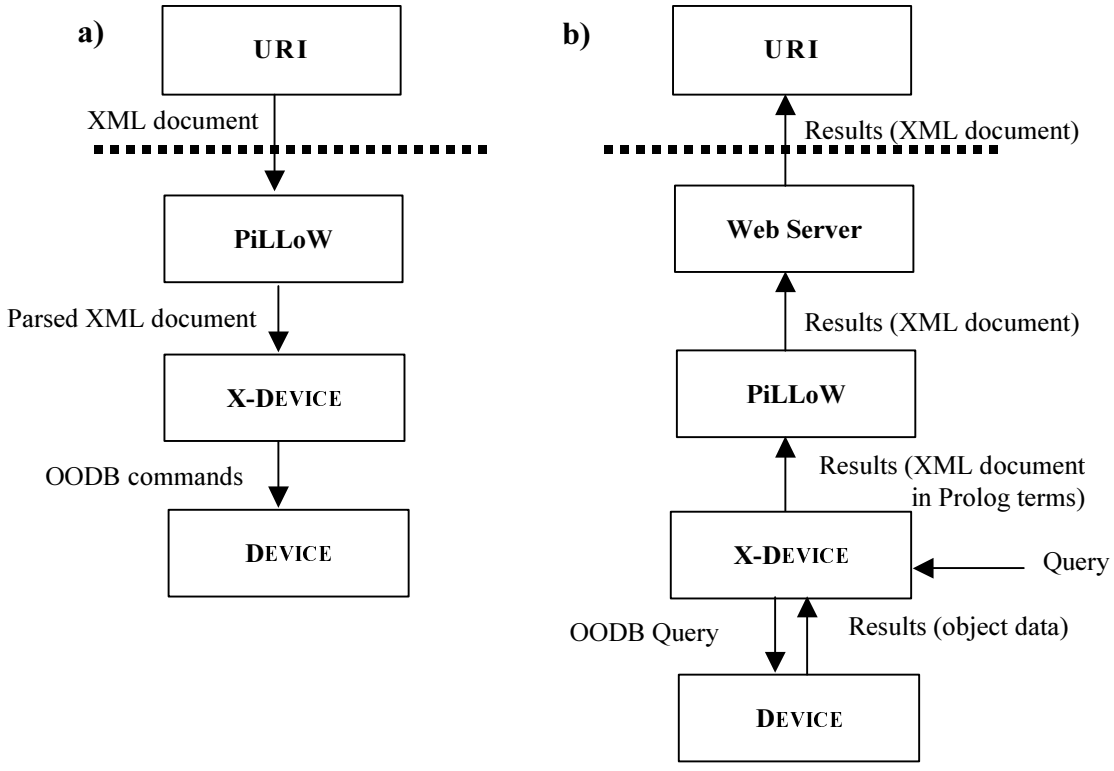


Figure 1. The architecture of the X-DEVICE system.

## Mapping XML Data onto the Object-Oriented Data Model

The X-DEVICE system incorporates XML documents with a schema described through a DTD into an object-oriented database. Figure 1 shows the architecture of the system. Specifically, XML documents (including DTD definitions) are fed into the system through the PiLLoW library (Cabeza & Hermenegildo, 2001), which parses them and transforms them into Prolog complex terms. DTD definitions are translated into an object database schema that includes classes and attributes, while XML data are translated into objects of the database. Generated classes and objects are stored within the underlying object-oriented database ADAM (Gray et al., 1992). It must be noticed that when an XML document lacks a DTD, then we can safely assume that one of the commercial XML editors, such as XML Spy (Altova), can generate a DTD for the XML document. In the future, we will include in the system a DTD induction algorithm.

Concerning the recently proposed XML Query Data Model (W3 Consortium, Dec 2001b), X-DEVICE currently maps a subset of the model's node types, namely: document, element, value, attribute, and node reference. These node types are mapped onto the object types presented in Figure 2. The mapping between a DTD and the object-oriented data model is done as follows:

- The *document node* type is represented by the `xml_doc` class and each document node is an instance of this class. The attributes of this class include the URI reference of the document and the OIDs of its root element nodes.
- *Element nodes* are represented as either object attributes or classes. More specifically:
  - If an element node has PCDATA content (without any attributes), it is represented as an attribute of the class of its parent element node. The name of the attribute is the same as the name of the element and its type is string.
  - If an element node has either a) children element nodes, or b) attribute nodes, then it is represented as a class that is an instance of the `xml_seq` meta-class. The attributes of the class include both the attributes of the element and the children element nodes. The types of the attributes of the class are determined as follows:
    - Simple character children element nodes correspond to attributes of string type.
    - Attribute nodes correspond to attributes of string type. Of course, this is an over-simplification because the string type is very broad. Actually, each of the different attribute type cases should be somehow maintained and handled within the OODB model. However, this cannot be done easily within the DTD framework, since DTDs do not provide typing information. Furthermore, typing information is important only for updating the XML document and not for just inserting it into the OODB and querying it, because it can be assumed that an XML validator has already taken care of typing and well-formdness details.
    - Children elements that are represented as objects correspond to object reference attributes.

- Special treatment needs the case when the content of an element is just PCDATA, but the element has attributes. In this case, the element is represented as a class, the element attributes as class attributes, and the element content as a string attribute called `content`.

```
class xml_doc
  attributes
    uri          (string, single, mandatory)
    children     (xml_elem, list, mandatory)
class xml_elem
  attributes
    alias        (attribute-xml_elem, set, optional)
    empty        (attribute, set, optional)
class xml_seq
  is_a           xml_elem
  attributes
    elem_ord     (attribute, list, optional)
    att_lst      (attribute, set, optional)
class xml_alt
  is_a           xml_elem
```

Figure 2. X-DEVICE object types for mapping XML documents.

- *Value nodes* are represented by the values of the object attributes. Currently, only strings and object references are supported since DTDs do not support data types. When XML Schema will be supported by X-DEVICE, the primitive data types of the OODB data model will be utilized.
- *Attribute nodes* are represented as object attributes. For the types of the attributes, the status is the same as for the value nodes. Attributes are distinguished from children elements through the `att_lst` meta-attribute.
- *Node references* are represented as object references.

Figure 5 shows an XML document that conforms to the TREE case DTD (Figure 3) and Figure 6 shows how this document is stored in X-DEVICE as a set of objects, whereas the class schema is shown in Figure 4. Notice that when a new element-object has exactly the same set of attribute values with an existing object (e.g. objects `0#author`, `5#book_alt1`), then it is not re-created for space efficiency. More examples of OODB schemata that are generated using our mapping scheme can be found in (X-DEVICE Web site).

There are more issues that a complete mapping scheme needs to address, except for the above mapping rules. First, elements in a DTD can be combined through either sequencing or alternation. *Sequencing* means that a certain element must include *all* the specified children elements with a specified order. This is handled by the above mapping scheme through the existence of multiple attributes in the class that represents the parent element, each for each child element of the sequence. The order is handled outside the standard OODB model by providing a meta-attribute (`elem_ord`) that specifies the correct ordering of the children elements. This meta-attribute is used when (either whole or a part of) the original XML document is reconstructed and returned to the user. The query language also uses it, as it will be shown later.

```
<!ELEMENT book (title, author+, section+)>
<!ELEMENT title (#PCDATA)>
<!ELEMENT author (#PCDATA)>
<!ELEMENT section (title, (p | figure | section)* )>
  <!ATTLIST section    id          ID     #IMPLIED
                       difficulty  CDATA  #IMPLIED>
<!ELEMENT p (#PCDATA)>
<!ELEMENT figure (title, image)>
  <!ATTLIST figure     width       CDATA  #REQUIRED
                       height      CDATA  #REQUIRED>
<!ELEMENT image EMPTY>
  <!ATTLIST image      source      CDATA  #REQUIRED>
```

Figure 3. The DTD of the TREE case.

```
xml_seq book
  attributes
    title          (string, single, mandatory)
    author         (string, list, mandatory)
    section        (section, list, mandatory)
  meta_attributes
    elem_ord       [title, author, section]
xml_seq section
  attributes
    title          (string, single, mandatory)
    section_alt1   (section_alt1, list, optional)
    id             (string, single, optional)
    difficulty     (string, single, optional)
  meta_attributes
    elem_ord       [title, section_alt1]
    att_lst        [id, difficulty]
    alias          [p-section_alt1, figure-section_alt1,section-
section_alt1]
xml_alt section_alt1
  attributes
    p              (string, single, optional)
    figure         (figure, single, optional)
    section        (section, single, optional)
xml_seq figure
  attributes
    title          (string, single, mandatory)
    image          (image, single, mandatory)
    width          (string, single, mandatory)
    height         (string, single, mandatory)
  meta_attributes
    elem_ord       [title, image]
    att_lst        [width, height]
xml_seq image
  attributes
    source         (string, single, mandatory)
  meta_attributes
    att_lst        [source]
```

Figure 4. The X-DEVICE class schema for the TREE case.

On the other hand, *alternation* means that *any* of the specified children elements can be included in the parent element. Alternation is also handled outside the standard OODB model by creating a new class for each alternation of elements, which is an instance of the `xml_alt` meta-class

and it is given a system-generated unique name. The attributes of this class are determined by the elements that participate in the alternation. The types of the attributes are determined as in the sequencing case. The structure of an alternation class may seem similar to a sequencing class, however the behavior of alternation objects is different, because they must have a value for exactly one of the attributes specified in the class (Figure 6).

```
<bib>
  <book year="1994">
    <title>TCP/IP Illustrated</title>
      <author><last>Stevens</last><first>W.</first></author>
      <publisher>Addison-Wesley</publisher>
      <price> 65.95</price>
  </book>
  <book year="1992">
    <title>Advanced Programming in the Unix environment</title>
    <author><last>Stevens</last><first>W.</first></author>
    <publisher>Addison-Wesley</publisher>
    <price>65.95</price>
  </book>
  <book year="2000">
    <title>Data on the Web</title>
    <author><last>Abiteboul</last><first>Serge</first></author>
    <author><last>Buneman</last><first>Peter</first></author>
    <author><last>Suciu</last><first>Dan</first></author>
    <publisher>Morgan Kaufmann Publishers</publisher>
    <price> 39.95</price>
  </book>
  <book year="1999">
    <title>The Economics of Technology and Content for Digital TV</title>
    <editor>
      <last>Gerbarg</last><first>Darcy</first>
      <affiliation>CITI</affiliation>
    </editor>
    <publisher>Kluwer Academic Publishers</publisher>
    <price>129.95</price>
  </book>
</bib>
```

Figure 5. A sample XML document conforming to the TREE case DTD.

The alternation class is always encapsulated in a parent element. The parent element class has an attribute with the system-generated name of the alternation class, which should be hidden from the user for querying the class. Therefore, a meta-attribute (`alias`) is provided with the aliases of this system-generated attribute, i.e. the names of the attributes of the alternating class.

Mixed content elements are handled similarly to alternation of elements. The only difference is that one of the alternative children elements can also be plain text (PCDATA), which is handled by creating a string attribute of the alternation class, called `content`.

Another issue that must be addressed is the mapping of the occurrence operators for elements, sequences and alternations. More specifically, these operators are handled as follows:

- The "star"-symbol (*) after a child element causes the corresponding attribute of the parent element class to be declared as an optional, multi-valued attribute.

- The "cross"-symbol (+) after a child element causes the corresponding attribute of the parent element class to be declared as a mandatory, multi-valued attribute.
- The question mark (?) after a child element causes the corresponding attribute of the parent element class to be declared as an optional, single-valued attribute.

```
object        14#bib
  instance    bib
  attributes
    book   [11#book,12#book,13#book]
object        10#book
  instance    book
  attributes
    year      '1994'
    title     'TCP/IP Illustrated
    book_alt1 [5#book_alt1]
    publisher 'Addison-Wesley'
    price     '65.95'

object        11#book
  instance    book
  attributes
    year      '1992'
    title     'Advanced Programming
               in the Unix
               environment'
    book_alt1   [5#book_alt1]
    publisher   'Addison-Wesley'
    price       '65.95'

object      12#book
  instance    book
  attributes
    year        '2000'
    title       'Data on the Web'
    book_alt1   [6#book_alt1,
                 7#book_alt1,
                 8#book_alt1]
    publisher   'Morgan Kaufmann
                 Publishers'
    price       '39.95'

object      13#book
  instance    book
  attributes
    year        '1999'
    title       'The Economics of
                 Technology and
                 Content for
                 Digital TV'
    book_alt1   [9#book_alt1]
    publisher   'Kluwer Academic
                 Publishers'
    price       '129.95'

object        5#book_alt1
  instance    book_alt1
  attributes
    author    0#author
    editor    ∅
object        6#book_alt1
  instance    book_alt1
  attributes
    author    1#author
    editor    ∅
object        7#book_alt1
  instance    book_alt1
  attributes
    author    2#author
    editor    ∅
object        8#book_alt1
  instance    book_alt1
  attributes
    author    3#author
    editor    ∅
object        9#book_alt1
  instance    book_alt1
  attributes
    author    ∅
    editor    4#editor

object         0#author
  instance     author
  attributes
    last       'Stevens'
    first      'W.'
object         1#author
  instance     author
  attributes
    last       'Abiteboul'
    first      'Serge'
object         2#author
  instance     author
  attributes
    last       'Buneman'
    first      'Peter'
object         3#author
  instance     author
  attributes
    last       'Suciu'
    first      'Dan'
object         4#editor
  instance     editor
  attributes
    last       'Gerbarg'
    first      'Darcy'
    affiliation 'CITI'
```

Figure 6. X-DEVICE representation for the XML document in Figure 5.

- Finally, the absence of any symbol means that the corresponding attribute should be declared as a mandatory, single-valued attribute.

The order of children element occurrences is important for XML documents, therefore the multi-valued attributes are implemented as lists and not as sets.

The combination of occurrence operators and encapsulated sequences of children elements calls for special treatment. For example, consider the following DTD declaration:

```
<!ELEMENT  a  (b, (c, d)*, e)>
```

The sequence `(c, d)` can appear multiple times inside the element `a`. However, the structure of a class is only finite. This case is handled by a system-generated sequencing class that includes the encapsulated elements `c` and `d`. The system-generated unique name of this class is also the name of the attribute of the parent element class. Finally, a meta-attribute with the aliases for the system-generated name exists, as it is the case for encapsulated alternating children elements (see above).

Empty elements are treated in the framework described above, depending on their internal structure. If an empty element does not have attributes, then it is treated as a PCDATA element, i.e. it is mapped onto a string attribute of the parent element class. The only value that this attribute can take is `yes`, if the empty element is present. If the empty element is absent then the corresponding attribute does not have a value. On the other hand, if an empty element has attributes, then is represented by a class. Finally, unstructured elements that have content `ANY` are not currently treated by X-DEVICE.

## The X-DEVICE Deductive Query Language

Users can query the stored XML documents using X-DEVICE, by: a) submitting the query through an HTML form, b) submitting an XML document that encapsulates the X-DEVICE query as a string, or c) entering the query directly in the text-based Prolog environment. In any of the above ways, the X-DEVICE query is finally forwarded to the X-DEVICE query processor, which translates it into the first-order language of DEVICE (Figure 1). The latter executes the query and returns the results to the X-DEVICE component. The results are then transformed into Prolog terms that represent XML data. These data structures are fed into the PiLLoW library, which returns them to the user in the form of an XML document.

In this section, we initially give a brief overview of the DEVICE deductive rule language, which serves as the basis for querying the stored XML data. More details about DEVICE can be found in (Bassiliades et al., 2000; Bassiliades, Vlahavas, Elmagarmid, & Houstis, 2001).

### *The Basic Deductive Query Language of DEVICE*

The syntax for X-DEVICE deductive rules is given in the Appendix. Rules are composed of condition and conclusion, whereas the condition defines a pattern of objects to be matched over the database and the conclusion is a derived class template that defines the objects that should be in the

database when the condition is true. The following rule defines that an object with attribute `end=Y` exists in class `path_from_one` if there is an object with OID `A` in class `arc` with attributes `start=1`, `end=Y`.

```
if A@arc(start=1,end:Y)
then path_from_one(end:Y)
```

Class `path_from_one` is a derived class, i.e. a class whose instances are derived from deductive rules. Only one derived class template is allowed at the THEN-part (head) of a deductive rule. However, there can exist many rules with the same derived class at the head. The final set of derived objects is a union of the objects derived by the two rules. For example, the transitive closure of the set of nodes reachable from node `1` is completed with the following (recursive) rule:

```
if P@path_from_one(end:Y) and
   A@arc(start:Y,end:Z\=1)
then path_from_one(end:Z)
```

The syntax of the basic DEVICE rule language is first-order. Variables can appear in front of class names (e.g. `P`, `A`), denoting OIDs of instances of the class, and inside the brackets (e.g. `Y`, `Z`), denoting attribute values (i.e. object references and simple values, such as integers, strings, etc). Conditions also can contain comparisons between attribute values, constants and variables (e.g. `end:Z\=1`). Negation is also allowed if rules are safe, i.e. variables that appear in the conclusion must also appear at least once inside a non-negated condition.

A query is executed in DEVICE by submitting the set of stratified rules (or logic program) to the system, which translates them into active rules and activates the basic events to detect changes at base data. Data then are forwarded to the rule processor through a discrimination network (much alike in a production system fashion). Rules are executed with fixpoint semantics (semi-naive evaluation), i.e. rule processing terminates when no more new derivations can be made. Derived objects are materialized and are either maintained after the query is over or discarded on user's demand. DEVICE also supports production rules, which have at the THEN-part one or more actions expressed in the procedural language of the underlying OODB (Gray et al., 1992).

The main advantage of the DEVICE system is its extensibility that allows the easy integration of new rule types as well as transparent extensions and improvements of the rule matching and execution phases. The current system implementation includes deductive rules for maintaining derived and aggregate attributes. Among the optimizations of the rule condition matching is the use of a RETE-like discrimination network, extended with re-ordering of condition elements for reducing time complexity and virtual-hybrid memories for reducing space complexity (Bassiliades & Vlahavas, 1997). Furthermore, set-oriented rule execution can be used for minimizing the number of inference cycles (and time) for large data sets (Bassiliades et al., 2000).

*Extending DEVICE for Querying XML Data*

The deductive rule language of DEVICE is extended with new constructs and operators in order to facilitate traversing and querying of the tree-structured XML data. The new constructs are implemented using second-order logic syntax (i.e. variables can range over class and attribute names) that has been introduced to DEVICE for integrating heterogeneous schemata (Bassiliades et al., 2001).

Both the new constructs and the second-order syntax are translated into a combination of: a) a set of first-order logic deductive rules, and/or b) a set of production rules that their conditions query the meta-classes of the OODB, they instantiate the second-order variables, and they dynamically generate first-order deductive rules.

Throughout this section, we will demonstrate the use of X-DEVICE for querying XML data using examples taken from the XML Query Use Cases proposed by the WWW consortium (W3 Consortium, Dec 2001c). Furthermore, we present the translation of the various new constructs to the basic DEVICE rule language. The general procedures for query translation can be found in (X-DEVICE Web site).

Path Expressions

X-DEVICE supports several types of path expressions into rule conditions. The simplest case is when all the steps of the path must be determined. This case can be handled by the basic mechanism of path expressions of DEVICE without any extension. The following example demonstrates fully determined path expressions, using the query Q3 of the SGML use case (Figure 7).

```
<!ELEMENT report (title, chapter+)>
<!ELEMENT title (#PCDATA | emph)*>
<!ELEMENT chapter (title, intro?, section*)>
  <!ATTLIST chapter    shorttitle CDATA #IMPLIED>
<!ELEMENT intro (para | graphic)+>
<!ELEMENT section (title, intro?, topic*)>
  <!ATTLIST section    shorttitle CDATA #IMPLIED
                       sectid ID #IMPLIED>
<!ELEMENT topic (title, (para | graphic)+)>
  <!ATTLIST topic      shorttitle CDATA #IMPLIED
                       topicid ID #IMPLIED>
<!ELEMENT para (#PCDATA | emph | xref)*>
  <!ATTLIST para       security (u | c | s | ts) "u">
<!ELEMENT emph (#PCDATA | emph)*>
<!ELEMENT graphic EMPTY>
  <!ATTLIST graphic    graphname ENTITY #REQUIRED>
<!ELEMENT xref EMPTY>
  <!ATTLIST xref       xrefid IDREF #IMPLIED>
```

Figure 7. The DTD of the SGML case.

*SGML Case - Q3*. Locate all paragraphs in the introduction of a section that is in a chapter that has no introduction (all "para" elements directly contained within an "intro" element directly contained in a "section" element directly contained in a "chapter" element. The "chapter" element must not directly contain an "intro" element).

The X-DEVICE version of the above query is the following:

```
if C@chapter(intro \= I, para.intro.section ϶ P)
then result(para:list(P))
```

Several features of X-DEVICE are demonstrated through this example. First of all, the path expressions are composed using dots between the "steps", which are attributes of the interconnected objects that represent XML document elements. The innermost attribute should be an attribute of "departing" class, i.e. `section` is an attribute of class `chapter`. Moving to the left, attributes belong to classes that represent their predecessor attributes. Notice that we have adopted a right-to-left order of attributes, contrary to the C-like dot notation that is commonly assumed, because we would like to stress out the functional data model origins of the underlying ADAM OODB (Gray et al., 1992). Under this interpretation the chained "dotted" attributes can be seen as function compositions.

The rules that contain path expressions are transformed into equivalent rules that contain only single-step path expressions, during the pre-compilation phase of DEVICE. The above rule is transformed into the following:

```
if C@chapter(intro \= I, section ϶ XX1) and
   XX1@section(intro:XX2) and
   XX2@intro(para ϶ P)
then result(para:list(P))
```

Variables `XX1` and `XX2` are generated by the system. Variables are instantiated through the '`:`' operator when the corresponding attribute is single-valued, and the '϶' operator when the corresponding attribute is multi-valued. Since multi-valued attributes are implemented through lists (ordered sequences) the '϶' operator guarantees that the instantiation of variables is done in the predetermined order stored inside the list.

The `list(P)` construct in the rule condition denotes that the attribute `para` of the derived class `result` is an attribute whose value is calculated by the aggregate function `list`. This function collects all the instantiations of the variable `P` and stores them under a strict order into the multi-valued attribute `para`. More details about the implementation of aggregate functions in DEVICE can be found in (Bassiliades et al., 2000).

Notice that the attribute `para` of the class `intro` is part of an encapsulated alternation; therefore, the `para` objects are accessible through an alternation class called `intro_alt1`. Consequently, the condition `XX2@intro(para ϶ P)` is actually expanded to the following expression, wherever it occurs in the above translation:

```
XX2@intro(intro_alt1 ϶ XX3) and XX3@intro_alt1(para:P)
```

Actually, the above expansion is performed in all the path expressions that involve elements included in an alternation. In the rest of the examples we will not further present this expansion to keep the presentation simple. □

The same query in XQuery is expressed as follows:

```
<result>
 {  for $c in //chapter
    where empty($c/intro)
    return $c/section/intro/para
 }
</result>
```

Comparing XQuery with X-DEVICE, we notice that in XQuery there are explicit functional-style looping constructs, while in X-DEVICE looping is implicit since it is a declarative DATALOG-like language that follows the semi-naive evaluation algorithm. Furthermore, XQuery has separate syntactical constructs for selecting and returning results, whereas XML results are built in a template-like manner. Notice that the (//) operator searches an element throughout the XML tree, whereas the (/) operator searches an element only among the children of the predecessor element. In general, XQuery has not a clear and simple syntax but follows multiple different paradigms. In contrast, the syntax of X- DEVICE follows a simple paradigm, that of a logic-like notation enhanced with OODB and general path extensions.

Another case in path expressions is when the number of steps in the path is determined, but the exact step name is not. In this case, a variable is used instead of an attribute name. This is demonstrated by the following example, which is a simplification of the question Q2 of the SEQ case (Figure 8): "In the Procedure section, what Instruments were used?"

Knowing that between instrument elements and the section content element there is one other element whose name is unknown, the above query in X-DEVICE looks like the following:

```
if S@section(section_title='Procedure',instrument.C.section_content ∋ I)
then result(instrument:list(I))
```

```
<!ELEMENT report (section*)>
<!ELEMENT section (section.title, section.content)>
<!ELEMENT section.title  (#PCDATA )>
<!ELEMENT section.content  (#PCDATA|anesthesia|prep|incision|action
                            |observation)*>
<!ELEMENT action ( (#PCDATA | instrument )* )>
<!ELEMENT prep ( (#PCDATA | action)* )>
<!ELEMENT incision ( (#PCDATA | geography | instrument)* )>
<!ELEMENT anesthesia (#PCDATA)>
<!ELEMENT observation (#PCDATA)>
<!ELEMENT geography (#PCDATA)>
<!ELEMENT instrument (#PCDATA)>
```

Figure 8. The DTD of the SEQ case.

Variable C is in the place of an attribute name, therefore it is a second-order variable, since it ranges over a set of attributes, and attributes are sets of things (attribute values). Deductive rules that contain second-order variables are always translated into a set of rules whose second-order variable has been instantiated with a constant. This is achieved by generating production rules, which query the meta-classes of the OODB, instantiate the second-order variables, and generate deductive rules with constants instead of second-order variables. The above deductive rule is translated into the following (simplified) production rule:

```
if section_content@xml_seq(elem_order ə C) and
   C@xml_seq(elem_order ə instrument)
then new_rule('if S@section(section_title='Procedure',section_content ə XX1)
                      and XX1@section_content(C ə XX2) and
                      XX2@C(instrument ə I)
                  then result(instrument:list(I))')
     => deductive_rule
```

Notice that variable `C` is now a first-order variable in the condition of the production rule, while the deductive rule generated by the action of the production rule has `C` instantiated. The above rule will actually produce two deductive rules, namely `C` is bound to `incision` and `action`. The result consists of the union of the results of the two deductive rules. The conditions of the two rules begin with the same condition element, which leads to an optimized execution due to the compact discrimination network that DEVICE produces. □

The most interesting case of path expressions is when some part of the path is unknown, regarding both the number and the names of intermediate steps. This is handled in X-DEVICE by using the "star" (*) operator in place of an attribute name. Such path expressions are called "generalized" or "general". The previous example can be re-written using the "star" (*) operator as:

```
if S@section(section_title='Procedure',instrument.*.section_content ə I)
then result(instrument:list(I))
```

The following set of rules, that represents the translation of the above rule, first establish the beginning of the path, starting from the "departing" class, then recursively navigate the elements of the XML tree, and finally terminate the path, either with a fixed element or with an element that does not have children. The last rule is a production rule that iterates over the discovered paths, which have been stored in a temporal class, and creates the corresponding deductive rules.

```
if section@xml_seq(elem_order ə section_content)
then tmp_elem1(cnd_elem:section_content,path_string:'section_content')

if XX1@tmp_elem1(cnd_elem:XX2 \= instrument,path_string:XX3) and
   XX2@xml_seq(elem_order ə XX5 \= instrument)
then tmp_elem1(cnd_elem:XX5,path_string:'XX5.XX3')

if XX1@tmp_elem1(cnd_elem:XX2 \= instrument,path_string:XX3) and
   XX2@xml_seq(elem_order ə instrument)
then tmp_elem2(path_string:'instrument.XX3')

if XX1@tmp_elem2(path_string:XX2)
then new_rule('if S@section(section_title='Procedure', XX2 ə I)
               then result(instrument:list(I))')
     => deductive_rule
```

Notice that the translation has been simplified for presentation purposes, by eliminating all rules and attributes that have to do with recursive elements, since the SEQ case DTD does not contain such elements. The translation procedure for rules containing generalized paths is more complex that the previous example, because the intermediate path may contain recursive elements, i.e. elements that contain other elements of the same type, as for example the element `section` of the TREE case (Figure 3). The translation of a rule that contains a recursive element is presented in the next example.

*TREE Case - Q2 (simple)*. Prepare a (flat) figure list for 'Book1', listing all the titles of the figures.

The above query is expressed in X-DEVICE as:

```
if B@book(title='Book1', title.figure.section*:T)
then figure1(title:list(T))
```

Notice that the result generates a class `figure1`, because class `figure` already exists. The above rule contains a path expression with a recursive element `section*`. This means that the search for figure titles will be performed in any nesting level of section elements. The nesting depth of section elements cannot be determined by just looking at the schema of the document, therefore the `section*` expression cannot be unfolded to multiple `section.section...section` paths of increasing length. The translation of the recursive element path of the above query is the following:

```
if B@book(title='Book1', title.figure.section:T)
then figure1(title:list(T))

if B@book(title='Book1', section ∋ XX1)
then tmp_elem1(cnd_obj:XX1)

if XX2@tmp_elem1(cnd_obj:XX1) and
   XX1@section(section ∋ XX3)
then tmp_elem1(cnd_obj:XX3)

if XX1@tmp_elem1(title.figure.cnd_obj:T)
then figure1(title:list(T))
```

The algorithm traverses down the tree of sections originating from a specific `book` object, copying the sections into the system-generated `tmp_elem1` class. For each of these sections the figure titles are retrieved and copied to the result. □

We notice here that XQuery (W3 Consortium, Dec 2001a) cannot express queries with general path expressions, i.e. queries with the star (*) operator, unless the star (*) operator lies at the beginning of the path. The following example demonstrates that.

*SGML Case - Q1*. Locate all paragraphs in the report (all "para" elements occurring anywhere within the "report" element) (Figure 7).

The above query is expressed in X- DEVICE as:

```
if R@report(para.* ∋ P)
then result(para:list(P)
```

whereas in XQuery the query is expressed as:

```
<result>
{    //report//para
  }
</result>
```

□

There are some cases that there are multiple solutions to a problem with a path expression and the user may require the solution that involves the shortest of the paths. For such cases, X-DEVICE

offers the "cross" (+) operator in the place of the "star" (*) operator. The next example demonstrates the use and translation of the "cross" (+) operator.

*SGML Case - Q10*. Locate the closest title preceding the cross-reference ("xref") element whose "xrefid" attribute is "top4" (the "title" element that would be touched last before this "xref" element when touching each element in document order).

The above query is expressed in X-DEVICE as:

```
if P@Element(title:T, ^xrefid.xref.+ = 'top4')
then result(title:T)
```

Notice that when the (^) symbol precedes an object attribute in X-DEVICE, then this designates an XML attribute, such as `^xrefid`. In the SGML DTD (Figure 7), there are many elements that encapsulate `title` as a child element, so the class name in the above rule is a second-order variable that iterates over all such elements. The following production rule finds all elements with a `title` sub-element and creates deductive rules with variable `Element` bound to constants.

```
if Element@xml_seq(elem_order ϶ title)
then new_rule('if P@Element(title:T, ^xrefid.xref.+ = 'top4')
                 then result(title:T)')
     => deductive_rule
```

The translation of each of the above deductive rules requires two rules: one for generating all possible results for all possible paths using the "star" (*) operator, and one for selecting the shortest of the paths, which is identified by an element that belongs to the result that it does not contain any other element that also belongs to the result.

```
if P@Element(title:T, ^xrefid.xref.*='top4')
then tmp_elem1(tmp_var1:T,tmp_obj:P)

if XX1@tmp_elem1(tmp_var1:T,tmp_obj:XX2) and
   not( XX3@tmp_elem1(tmp_obj:XX4\=XX2) and
        XX2@Element(* ϶ XX4) )
then result(title:T)
```

□

Finally, the user should be able to perform arbitrary text search within an XML document without having to worry about the internal structure of the document. X-DEVICE offers the (↑) operator that flattens an element returning a string consisting of all its sub-element contents (tags are not included). Then the user is able to perform text search inside this string. The following example demonstrates this operator.

*SGML Case - Q8a*. Locate all sections with a title that has "is SGML" in it (all "section" elements that contain a "title" element that has the consecutive characters "is SGML" in its content). The string can be interrupted by sub-elements (Figure 7).

The above query is expressed in X-DEVICE as:

```
if S@section(title↑ $ 'is SGML')
then result(section:list(S))
```

The flattening operator (↑) is placed after the name of the element to be flattened. This operator gets translated using two rules:

```
if S@section(*.title ϶ XX1)
then tmp_elem1(tmp_var1:S,tmp_val:string(XX1))

if XX1@tmp_elem1(tmp_var1:S,tmp_val $ 'is SGML')
then result(section:list(S))
```

The first rule collects the contents of all sub-elements of `title`, using a "star" (*) operator. All these contents are stored in an attribute of a temporary class, through the `string` aggregate function. This aggregate function collects all the (string) values of variable `XX1` and concatenates them together. The second rule searches inside the result string using the string search (`$`) operator of X-DEVICE. □

Ordering Expressions

X-DEVICE supports expressions that query an XML tree based on the ordering of elements. The implementation of such operators is based on the fact that multi-valued attributes are implemented using lists in which the order of stored values is fixed. The translation of ordering expressions requires multiple steps to preserve their semantics, especially when there are multiple such expressions in a single rule.

When a rule contains exactly one ordering expression, then the original rule is transformed into three rules that:

1. gather all the results,
2. restrict to as many results as the ordering expression requires, and
3. iterate over the correct results continuing with rest of the rule condition.

The following example demonstrates the use and translation of a simple ordering expression.

*SEQ Case - Q2*. In the Procedure section, what are the first two Instruments to be used? (Figure 8)

The above query is expressed in X-DEVICE as:

```
if S@section(section_title='Procedure',instrument.*.section_content ϶=<2 I)
then result(instrument:list(I))
```

The ϶=<2 operator is an absolute numeric ordering expression that returns the first two elements of the corresponding list-attribute. More such ordering expressions exist that are implemented accordingly. The translation of this expression is the following:

```
if S@section(section_title='Procedure',instrument.*.section_content ϶ XX1)
then tmp_elem1(tmp_obj:list(XX1))

if XX3@tmp_elem1(tmp_obj:XX1) and
   prolog{select_sub_list('=<'(2),XX1,XX2)}
then tmp_elem2(tmp_obj:XX2)

if XX1@tmp_elem2(tmp_obj ϶ I)
then result(instrument:list(I))
```

□

If a rule contains more than one ordering expressions, then the original rule is transformed into as many rules as the list operators. Furthermore, there is a shortcut notation for multiple absolute numeric ordering expressions, which is demonstrated with the following example:

*SGML Case - Q4*. Locate the second paragraph in the third section in the second chapter (the second "para" element occurring in the third "section" element occurring in the second "chapter" element occurring in the "report") (Figure 7).

The above query is expressed in X-DEVICE as:

```
if R@report(para2.section3.chapter2:P)
then result(para:P)
```

The above shortcut notation is translated into a sequence of multiple ordering expressions:

```
if R@report(chapter ϶2 XX1) and
   XX1@chapter(section ϶3 XX2) and
   XX2@section(para ϶2 P)
then result(para:P)
```

The operator $϶_n$ (or $϶_{=n}$) returns the n-th element in a sequence. The translation of the above rule is:

```
if R@report(chapter ϶2 XX1)
then tmp_elem1(tmp_obj1:list(XX1))

if XX3@tmp_elem1(tmp_obj1:XX1) and
   XX1@chapter(section ϶3 XX2)
then tmp_elem2(tmp_obj2:list(XX2))

if XX3@tmp_elem2(tmp_obj2 ϶ XX2) and
   XX2@section(para ϶2 P)
then result(para:P)
```

□

The translation of relative ordering expressions follows the same general procedure with absolute numeric ones. However, the combination of such expressions with alternation classes calls for a special treatment. This will be better explained through the following example:

*SEQ Case - Q5*. What happened between the first Incision and the second Incision? (Figure 8)

The above query is expressed in X-DEVICE as:

```
if S@section(incision.section_content ϶1 I1) and
   S@section(incision.section_content ϶2 I2) and
   S@section(H.section_content ϶between(I1,I2) W)
then result(happened:list(W))
```

The operator `between(I1,I2)` is a relative ordering expression that returns all elements in a sequence after the one with an OID identified by the instantiations of the variable `I1` and before the ones with OID `I2`. The problem here arises from the fact that the `section_content` element has a mixed content; therefore, all its children elements (including `incision`) are connected to element `section_content` indirectly through the class `section_content_alt1`. Therefore, the attribute

`section_content_alt1` of class `section_content` contains objects of class `section_content_alt1` and not objects of any of the classes of its children elements, such as `incision`, `action`, etc. Thus, the implementation of the operator `between(I1,I2)` must take into account that the OIDs of the children elements of `section_content` do not co-exist in the same list attribute.

The solution to the above problem is to replace all the variables that depend on the relative ordering expression, namely `I1`, `I2` and `W`, with new variables that represent their parent elements of the corresponding alternation class `section_content_alt1`. Furthermore, the parts of the condition that contain these variables must be transformed using the equivalent alternation classes. The above example is transformed as follows:

```
if S@section(section_content_alt1.section_content ϶1 XX1) and
   XX1@section_content_alt1(incision ϶ I1) and
   S@section(section_content_alt1.section_content ϶2 XX2) and
   XX2@section_content_alt1(incision ϶ I2) and
   S@section(section_content_alt1.section_content ϶between(XX1,XX2) XX3) and
   XX3@section_content_alt1(H ϶ W)
then result(happened:list(W))
```

□

In some cases, the relative ordering expression coexists with an absolute numeric ordering expression, which is called a complex ordering expression. In these cases, the rule with the two ordering expressions is cut down into two separate rules, each one containing one single ordering expression. The following example demonstrates this:

*SEQ Case - Q3*. What Instruments were used in the first two Actions after the second Incision? (Figure 8)

The above query is expressed in X-DEVICE as:

```
if S@section(incision.section_content ϶2 I) and
   S@section(action.section_content ϶{after(I), =<2} A) and
   A@action(instrument ϶ I2)
then result(instrument:list(I2))
```

The operator ϶{after(I),=<2} is a complex ordering expression that consists of the relative ordering expression `after(I)` followed by the absolute numeric ordering expression `=<2`. The translation is:

```
if S@section(incision.section_content ϶2 I) and
   S@section(action.section_content ϶after(I) XX1)
then tmp_elem1(tmp_obj:list(XX1))

if XX1@tmp_elem1(tmp_obj ϶=<2 A) and
   A@action(instrument ϶ I2)
then result(instrument:list(I2))
```

□

Exporting Results

So far, only the querying of existing XML documents through deductive rules has been discussed. However, it is important that the results of a query can be exported as an XML document. This can be performed in X-DEVICE by using some directives around the conclusion of a rule that defines the top-level element of the result document.

When the rule processing procedure terminates, X-DEVICE employs an algorithm that begins with the top-level element designated with one of these directives and navigates recursively all the referenced classes constructing a result in the form of an XML tree-like document. The procedure for answering an X-DEVICE query consists of compiling and running the query and then constructing the XML result document along with its DTD. Rule compilation and evaluation are not described here since they are actually a part of the DEVICE system. However, what should be mentioned is that the object schema of the derived classes is determined during the compilation phase, which returns the name of the top-element class of the result document.

The following example demonstrates how XML documents (and DTDs) are constructed in X-DEVICE for exporting them as results.

*XMP Case - Q1*. List books published by Addison Wesley after 1991, including their year and title (Figure 9).

The above simple query is expressed in X-DEVICE as:

```
if B@book(title:T,publisher='Addison Wesley',year:Y>1991)
then xml_result(book1(title:T,year:Y))
```

The keyword `xml_result` is a directive that indicates to the query processor that the encapsulated derived class (`book1`) is the answer to the query. This is especially important when the query consists of multiple rules. In order to build an XML tree as a query result, the objects that correspond to the elements must be constructed incrementally in a bottom-up fashion, i.e. first the simple elements that are towards the leaves of the tree are generated and then they are combined into more complex elements towards the root of the tree.

```
<!ELEMENT bib  (book* )>
<!ELEMENT book  (title,  (author+ | editor+ ), publisher, price )>
  <!ATTLIST book  year CDATA  #REQUIRED >
<!ELEMENT author  (last, first )>
<!ELEMENT editor  (last, first, affiliation )>
<!ELEMENT title  (#PCDATA )>
<!ELEMENT price  (#PCDATA )>
<!ELEMENT last  (#PCDATA )>
<!ELEMENT first  (#PCDATA )>
<!ELEMENT affiliation  (#PCDATA )>
<!ELEMENT publisher  (#PCDATA )>
```

Figure 9. The DTD of the XMP case.

The above query produces a forest of book elements, with the following DTD:

```
<!DOCTYPE book1 [
  <!ELEMENT book1 (title, year)>
```

```
]>
```

If a tree is wanted instead, then we should use one more rule to produce a top-level element (`bib1`) to wrap the `book1` elements:

```
if B@book(title:T,publisher='Addison Wesley',year:Y>1991)
then book1(title:T,year:Y)

if B@book1
then xml_result(bib1(book1:list(B)))
```

with the following DTD:

```
<!DOCTYPE bib1 [
  <!ELEMENT bib1 (book1)*>
  <!ELEMENT book1 (title, year)>
]>
```

Notice how the `list` aggregate function is used to construct XML elements with multiple children. In fact this kind of query is so common that X-DEVICE offers the following wrapping construct that is translated into the above set of rules:

```
if B@book(title:T,publisher='Addison Wesley',year:Y>1991)
then xml_result(bib1(book1(title:T,year:Y)))
```

□

One of the most important advantages of using a logic-like language for querying XML data is the ability of X- DEVICE to recursively query XML documents and construct a hierarchic document of arbitrary depth from flat structures stored in a database.

*PARTS Case - Q1*. Convert a flat list of parts that contain one another (Figure 10) to a tree of parts (Figure 11). In the result document, part containment is represented by containment of one `<part>` element inside another. Each part that is not part of any other part should appear as a separate top-level element in the output document.

The above query is expressed in X-DEVICE as:

```
if P@part(^partid:ID,^name:Name)
then part1(^partid:ID,^name:Name)

if PP1@part1(^partid:ID) and
   P1@part(^partid:ID,^parent:Parent) and
   PP2@part1(^partid:Parent)
then PP2@part1(part:list(PP1))

if P1@part1 and
   not P2@part1(part ϶ P1)
then xml_result(parttree(part1:list(P1)))
```

```
<!ELEMENT partlist (part*)>
<!ELEMENT part EMPTY>
<!ATTLIST part     partid  CDATA  #REQUIRED
                   partof  CDATA  #IMPLIED
                   name    CDATA  #REQUIRED>
```

Figure 10. The DTD of the PARTS case.

The above set of rules recursively iterates through the hierarchies implicit in the flat part list and transforms them into explicit complex object references in the recursive `part1` element. When the above algorithm finishes, the last rule makes top-level elements those that are contained within another part in he hierarchy.

```
<!ELEMENT parttree (part*)>
<!ELEMENT part (part*)>
<!ATTLIST part     partid  CDATA  #REQUIRED
                   name    CDATA  #REQUIRED>
```

Figure 11. The DTD for a tree of parts.

The same query in XQuery is expressed as:

```
define function one_level (element $p) returns element
{   <part partid={ $p/@partid }
          name={ $p/@name } >
{           for $s in document("data/parts-data.xml")//part
            where $s/@partof = $p/@partid
            return one_level($s)
        }
    </part>
}

<parttree>
{   for $p in document("data/parts-data.xml")//part[empty(@partof)]
    return one_level($p)
  }
</parttree>
```

The complexity of the above query lies in the fact that XQuery is functional and therefore requires explicit looping constructs, whereas the declarative, implicit loops of X-DEVICE allow a more clear and concise syntax. Furthermore, XQuery constructs the result in heterogeneous ways, i.e. either using templates or functions, which complicate things for the naive user, whereas X-DEVICE uses the simple notion of defining derived classes that can be referenced by other classes through object referencing. □

Another directive for constructing XML documents is `xml_sorted`, which is similar to `xml_result` and is used for sorting the elements of the result according to a group of element values specified in the rule head. The following example demonstrates the use and translation of `xml_sorted`:

*XMP Case - Q7*. List the titles and years of all books published by Addison Wesley after 1991, in alphabetic order (Figure 9).

```
if B@book(title:T,publisher='Addison Wesley',year:Y>1991)
then xml_sorted([T],bib1(book1(title:T,year:Y)))
```

Notice that the sorting is done on the values of the `T` variable, namely the book titles. This directive is easily translated into two rules:

```
if B@book(title:T,publisher='Addison Wesley',year:Y>1991)
then book1(title:T,year:Y)

if XX1@book1(title:T,year:Y)
```

```
then xml_result(bib1(book1:ord_list(XX1-[T])))
```

The first rule creates the inner class (`book1`), while the second one creates the outer class (`bib1`) and groups inside the single instance of the outer class all the instances of the inner class by copying them to a list attribute (`book1`), sorted by the values of the `T` variable. The expression `ord_list` is another aggregate function of X-DEVICE that, similarly to `list`, collects all the instances of its argument (variable `XX1`) into a list, sorting them simultaneously according to the values of the list that follows the argument (`[T]`). This function can be used independently of the `xml_sorted` construct, which means that the user can sort sibling elements anywhere in the result tree. □

Finally, the user can choose to avoid producing deep XML documents, even if the result contains elements that contain children elements, by using the `shallow_result` directive. This directive returns an XML document with two levels of nesting: the first contains only the root element of the result tree, and the second level contains elements without children but only with attributes (if any). The following example demonstrates this:

*REF Case - Q2*. Find Joe's children (Figure 12). Notice that the parent-child relationship is represented through the recursive element `person`.

```
<!ELEMENT census (person*)>
<!ELEMENT person (person*)>
  <!ATTLIST person   name    ID      #REQUIRED
                     spouse  IDREF   #IMPLIED
                     job     CDATA   #IMPLIED >
```

Figure 12. The DTD of the REF case.

The above query is expressed in X-DEVICE as:

```
if P@person(^name='Joe',person ϶ Ch)
then result(person:list(Ch))

if P@person(^name='Joe',^spouse:S) and
   P1@person(^name=S,person ϶ Ch)
then shallow_result(result(person:list(Ch)))
```

Notice that although both rules refer to the derived class `result`, only one of them contains the `shallow_result` directive. However, this is not a strict language rule; it does not matter if several rules contain the `shallow_result` or any other result directive, as long as the following constraints are satisfied:

- Only one type of result directive is allowed in the same query.
- Only one derived class is allowed at the result.

The DTD for the above set of rules is the following:

```
<!DOCTYPE result [
  <!ELEMENT result (person*)>
  <!ELEMENT person EMPTY>
  <!ATTLIST person     name    ID      #REQUIRED
                       spouse  IDREF   #IMPLIED
                       job     CDATA   #IMPLIED >
]>
```

although in the REF DTD the `person` element had sub-elements (Figure 12). □

Alternative Attribute Expressions

It has already been mentioned that X-DEVICE path expressions can contain steps that refer either to normal XML elements or attribute elements, using the (^) symbol as a prefix before the name. There are two more types of special attributes that can be used in X-DEVICE, namely the empty element attributes and the system attributes. The following example demonstrates the use of both special attributes.

*XMP Case - Q6*. For each book, list the title and first two authors, and an empty "et-al" element if the book has additional authors (Figure 9).

The above query is expressed in X-DEVICE as:

```
if B@book(title:T,author ∋=<2 A)
then book1(title:T,author:list(A))

if B1@book1(title:T) and
   B@book(title:T,author:A) and
   prolog{length(A,L), L>2}
then B1@book1(∅et_al)
```

The second rule above is a derived-attribute rule (Bassiliades et al., 2000), which creates a new `et_al` attribute for the class `book1`. This new attribute is actually an empty element prefixed by the (∅) symbol. Empty attributes are actually handled as strings with value `yes` if they are present. The derived-attribute rule defines that only objects of class `book1` that satisfy the condition will have such an attribute. The rest will have no value for this attribute, which means that no such element will appear at the result.

Another noticeable feature of both DEVICE and X-DEVICE in the above derived-attribute rule is the use of arbitrary Prolog goals inside the condition of rules. In this way the system can be extended with several features, which are outside of the language and therefore cannot be optimized.

Finally, notice that the derived-attribute rule iterates over all books that have been copied to the `book1` class retrieving their titles, and then matches the titles with those of the original `book` objects. If titles are not unique, however, this can produce wrong results. A safer way to do this would be to store within the `book1` objects the OID of the original book objects, which are unique. However, theses OIDs should not really be part of the query result; their purpose is merely auxiliary. This can be achieved by prefixing the auxiliary attribute with the (`!`) symbol, which indicates that it is a system attribute. Using a system attribute, the above query is now expressed as follows:

```
if B@book(title:T,author ∋=<2 A)
then book1(title:T,author:list(A),!original:B)

if B1@book1(!original:B) and
   B@book(author:A) and
   prolog{length(A,L), L>2}
then B1@book1(∅et_al)
```

The `original` attribute is a system attribute that will not appear at the final XML result. This is achieved by simply not placing the attribute name in the `elem_order` class attribute. Its functionality is to keep track (in `book1` objects) of the original `book` objects so that their XML attributes can be preserved through the first rule, for using them in the second rule. □

## Conclusions and Future Work

In this chapter, we have considered the problem of storing an XML document into an OODB by automatically mapping the schema of the XML document (DTD) to an object schema and XML elements to database objects. Our approach maps elements either as classes or attributes based on the complexity of the elements of the DTD, without loosing the relative order of elements in the original document.

Furthermore, we have provided a deductive rule query language for expressing queries over the stored XML data. The deductive rule language has certain constructs (such as second-order variables, general path and ordering expressions) for traversing tree-structured data that were implemented by translating them into first-order deductive rules. The translation scheme is mainly based on the querying of meta-data (meta-classes) about database objects. Comparing X-DEVICE with the XQuery query language, it seems that the high-level, declarative syntax of X-DEVICE allows users to express everything that XQuery can express, in a more compact and comprehensible way, with the powerful addition of general path expressions, fixpoint recursion and second-order variables.

Users can also express complex XML document views using X-DEVICE, a fact that can greatly facilitate customizing information for e-commerce and/or e-learning. Furthermore, the X-DEVICE system offers an inference engine that supports multiple knowledge representation formalisms (deductive, production, active rules, as well as structured objects), which can play an important role as an infrastructure for the impending semantic Web (W3 Consortium, Nov 2001). Production rules can also be used for updating an XML document inside the OODB, a feature not yet touched upon the XQuery initiative. However, the study of using production rules for updating XML documents is outside the scope of this chapter and is a topic of future research.

Among our plans for further developing X-DEVICE is the definition of an XML-compliant syntax for the rule/query language based on the upcoming RuleML initiative (Boley, Tabet, & Wagner, 2001). Furthermore, we plan to extend the current mapping scheme to documents that comply with XML Schema (W3 Consortium, May 2001).

*Acknowledgements*

Part of the work presented in this chapter was partially financially supported by the European Commission under the IST No 12503 Project "KOD - Knowledge on Demand" through the Information Society Technologies Programme (IST).

The first author is supported by a scholarship from the Greek Foundation of State Scholarships (F.S.S. - I.K.Y.).

## Appendix: X-DEVICE Syntax

```
<rule> ::= if <condition> then <consequence>
<condition> ::= <inter-object>
<consequence> ::= {<action> | <conclusion> | <derived_attribute_template>}
<inter-object> ::= <condition-element> ['and' <inter-object>]
<inter-object> ::= <inter-object> 'and' <prolog_cond>
<condition-element> ::= ['not'] <intra-object>
<intra-object> ::= [<inst_expr>'@']<class_expr>['('<attr-patterns>')']
<attr-patterns> ::= <attr-pattern>[','<attr-patterns>]
<attr-pattern> ::= <attr-expr>['.'<path_expr>] {':'<variable>
                     | <predicates>
                     | ':'<variable> <predicates>
                     | <list-operator> <variable>}
<path_expr> ::= <nt-attr-expr> ['.'<path_expr>]
<attr-expr> ::= {<nt-attr-expr>|<t-attr>|<normal-attr>'↑'}
<nt-attr-expr> ::= <nt-attr>[{'*'|<integer>}]
<nt-attr-expr> ::= {'*'|'+'}
<nt-attr> ::= {<normal-attr>|<system-attr>}
<t-attr> ::= {<xml-attr>|<empty-attr>}
<normal-attr> ::= <attr>
<system-attr> ::= '!'<attr>
<xml-attr> ::= '^'<attr>
<empty-attr> ::= '∅'<attr>
<attr> ::= {<attribute>|<variable>}
<predicates> ::= <rel-operator> <value> [{ '&' | ';' } <predicates>]
<predicates> ::= <set-operator> <set>
<rel-operator> ::= '=' | '>' | '>=' | '=<' | '<' | '\=' | '$'
                   | <date-operator>
<date-operator> ::= '$'{'y'|'m'|'d'}
<set-operator> ::= '⊂' | '⊄' | '⊆' | '∈' | '∉' | '⊃' | '\⊃' | '⊇'
<list-operator> ::= 'ɔ' | '\ɔ'
<list-operator> ::= 'ɔ'<order_expr>
<order_expr> ::= {<abs_order>|<rel_order>|'{'<rel_order>','<abs_order>'}'}
<abs_order> ::= <rel-operator><integer> | <integer>
<rel_order> ::= { 'before' | 'after' }'('<variable>')'
<rel_order> ::= 'between' '('<variable>','<variable>')'
<value> ::= <constant> | <variable> | <arith_expr>
<set> ::= '['<constants>']'
<prolog_cond> ::= 'prolog' '{'<prolog_goal>'}'
<action> ::= <prolog_goal>
<conclusion> ::= <derived class template>
<conclusion> ::= {'xml_result' | 'shallow_result'}'('<elem_expr>')'
<conclusion> ::= {'xml_sorted' | 'shallow_sorted'}'('[<group_list>
                                        ['-'<order_list>]',']<elem_expr>')'
<elem_expr> ::= <derived_class_template>
<elem expr> ::= <derived class>'('<derived class template>')'
<derived_class_template> ::= <derived_class>'('<templ-patterns>')'
<derived_attribute_template> ::= <variable>'@'{<class>}
                                 '('<templ-patterns>')'
<templ-patterns> ::= <templ-pattern> [',' <templ-pattern>]
<templ-pattern> ::= {<normal-attr>|<system-attr>|<xml-attr>}':'
                     {<value> | <aggregate_expr>}
<templ-pattern> ::= <empty-attr>
<aggregate_expr> ::= <aggregate_function>'('<variable>')'}
```

```
<aggregate_expr> ::= 'ord_list('<variable>['-'<group_list>
                                          ['-'<order_list>]]')'
<aggregate_function> ::= 'count'|'sum'|'avg'|'max'|'min'|'list'|'string'
<group_list> ::= '['<variable>[','<variable>]']'
<order_list> ::= '['<ord_symbol>[','<ord_symbol>]']'
<ord_symbol> ::= {'<' | '>'}
<inst_expr> ::= {<variable>|<class>}
<class_expr> ::= {<variable>|<class>|<inst_expr>'/'<class>}
<class> ::= an existing class or meta-class of the OODB schema
<derived_class> ::= an existing derived class or a non-existing base class of the OODB schema
<attribute> ::= an existing attribute of the corresponding OODB class
<prolog_goal> ::= an arbitrary Prolog/ADAM goal
<constants> ::= <constant>[','<constants>]
<constant> ::= a valid constant of an OODB simple attribute type
<variable> ::= a valid Prolog variable
<arith_expr> ::= a valid Prolog arithmetic expression
```

# References


Abiteboul, S., Cluet, S., Christophides, V., Milo, T., Moerkotte, G., Siméon, J. (1997a). Querying Documents in Object Databases. *International Journal on Digital Libraries*, *1* (1), 5-19.

Abiteboul, S., Quass, D., McHugh, J., Widom, J., & Wiener, J. L. (1997b). The Lorel Query Language for Semistructured Data. *International Journal on Digital Libraries*, *1* (1), 68-88.

Altova. (n.d.). *XML Spy*, Retrieved January 10, 2002 from http://www.xmlspy.com

Bassiliades, N. & Vlahavas, I. (1997). Processing Production Rules in DEVICE, an Active Knowledge Base System. *Data & Knowledge Engineering*, *24* (2), 117-155.

Bassiliades, N., Vlahavas, I., & Elmagarmid, A. K. (2000). E-DEVICE: An extensible active knowledge base system with multiple rule type support. *IEEE Transactions on Knowledge and Data Engineering*, *12* (5), 824-844.

Bassiliades, N., Vlahavas, I., Elmagarmid, A. K., & Houstis, E. N. (2001). InterBase$^{KB}$: Integrating a Knowledge Base System with a Multidatabase System for Data Warehousing. *IEEE Transactions on Knowledge and Data Engineering*, (to appear).

Boehm, K., Aberer, K., Neuhold, E. J., & Yang, X. (1997). Structured Document Storage and Refined Declarative and Navigational Access Mechanisms in HyperStorM. *Very Large Databases (VLDB) Journal*, *6* (4), 296-311.

Boley, H., Tabet, S., & Wagner, G. (2001). Design Rationale of RuleML: A Markup Language for Semantic Web Rules. *Proceedings of the International Semantic Web Working Symposium*, 381-402. Retrieved January 10, 2002 from http://www.dfki.uni-kl.de/ruleml/

Buneman, P., Davidson, S. B., Hillebrand, G. G., & Suciu, D. (1996). A Query Language and Optimization Techniques for Unstructured Data. *Proceedings of the ACM SIGMOD Conference*, 505-516.

Buneman, P., Fernandez, M., & Suciu, D. (2000). UnQL: A query language and algebra for semistructured data based on structural recursion. *Very Large Databases (VLDB) Journal*, *9* (1), 76-110.

Cabeza, D., & Hermenegildo, M. V. (2001). Distributed WWW Programming using (Ciao-) Prolog and the PiLLoW library. *Theory and Practice of Logic Programming, 1* (3), 251-282.

Cattell, R. G. G. (1994). *The object database standard: ODMG-93*. Morgan Kaufmann Publishers.

Chamberlin, D., Robie, J., & Florescu, D. (2000). Quilt: an XML Query Language for Heterogeneous Data Sources. *Proceedings of the International Workshop on the Web and Databases*, 53-62.

Chung, T.-S., Park, S., Han, S.-Y., & Kim, H.-J. (2001). Extracting Object-Oriented Database Schemas from XML DTDs Using Inheritance. *Proceedings of the International Conference on Electronic Commerce and Web Technologies (EC-Web)*. Springer-Verlag, LNCS *2115*, 49-59.

Deutsch, A., Fernandez, M., Florescu, D., Levy, A., & Suciu, D. (1999a). A Query Language for XML. *Computer Networks*, *31* (11-16), 1155-1169.

Deutsch, A., Fernandez, M. F., & Suciu, D. (1999b). Storing Semistructured Data with STORED. *Proceedings of the ACM SIGMOD Conference*, 431-442.

Diaz, O., & Jaime, A. (1997). EXACT: An Extensible Approach to Active Object-Oriented Databases. *Very Large Databases (VLDB) Journal*, *6* (4), 282-295.

Florescu, D., & Kossmann, D. (1999). Storing and Querying XML Data using an RDMBS. *IEEE Data Engineering Bulletin*, *22* (3), 27-34.

Goldman, R., & Widom, J. (1997). DataGuides: Enabling Query Formulation and Optimization in Semistructured Databases. *Proceeding of the International Conference on Very Large Databases (VLDB)*, 436-445.

Gray, P. M. D., Kulkarni, K. G., & Paton, N. W. (1992). *Object-Oriented Databases, A Semantic Data Model Approach*, London:Prentice Hall.

Hosoya, H., & Pierce, B. (2000). XDuce: A Typed XML Processing Language. *Proceedings of the International Workshop on Web and Database*, 111-116.

Klettke, M., & Meyer, H. (2000). XML and Object-Relational Database Systems - Enhancing Structural Mappings Based on Statistics. *Proceedings of the International Workshop on Web and Database*, 63-68.

Lakshmanan, L. V. S., Sadri, F., & Subramanian, I. N. (1996). A Declarative Language for Querying and Restructuring the Web. *Proceedings of the International Workshop on Research Issues in Data Engineering - Interoperability of Nontraditional Database Systems (RIDE-NDS)*, 12-21

Lucie Xyleme (2001). A Dynamic Warehouse for XML Data of the Web. *IEEE Data Engineering Bulletin*, *24* (2), 40-47.

Ludäscher, B., Himmeröder, R., Lausen, G., May, W., & Schlepphorst, C. (1998). Managing Semistructured Data with FLORID: A Deductive Object-Oriented Perspective. *Information Systems*, *23* (8), 589-613.

May, W. (2001). XPathLog: A Declarative, Native XML Data Manipulation Language. *Proceedings of the International Database Engineering and Application Symposium (IDEAS)*, 123-128.

McHugh, J., Abiteboul, S., Goldman, R., Quass, D., & Widom, J. (1997). Lore: A Database Management System for Semistructured Data, *ACM SIGMOD Record*, *26* (3), 54-66.

Naughton, J. F., DeWitt, D. J., Maier, D., Aboulnaga, A., Chen, J., Galanis, L., et al. (2001). The Niagara Internet Query System, *IEEE Data Engineering Bulletin*, *24* (2), 27-33.

Nishioka, S., & Onizuka, M. (2001). Mapping XML to Object Relational Model. *Proceedings International Conference on Internet Computing*, 171-177.

Ozsu, M. T., Iglinski, P., Szafron, D., El-Medani, S., & Junghanns, M. (1997). An Object-Oriented SGML/HyTime Compliant Multimedia Database Management System. *Proceedings of the ACM Multimedia Conference*, 239-249.

Renner, A. (2001). XML Data and Object Databases: A Perfect Couple?. *Proceedings of the International Conference on Data Engineering*, 143-148.

Robie, J., Lapp, J., & Schach, D. (n.d.) XML Query Language (XQL). Retrieved January 10, 2002, from http://www.w3.org/TandS/QL/QL98/pp/xql.html

Schmidt, A., Kersten, M. L., Windhouwer, M., & Waas, F. (2000). Efficient Relational Storage and Retrieval of XML Documents. Proceedings of the *International Workshop on the Web and Databases*, 47-52.

Shanmugasundaram, J., Tufte, K., Zhang, C., He, G., DeWitt, D. J., & Naughton, J. F. (1999). Relational Databases for Querying XML Documents: Limitations and Opportunities. *Proceedings of the International Conference on Very Large Databases (VLDB)*, 302-314.

Shimura, T., Yoshikawa, M., & Uemura, S. (1999). Storage and Retrieval of XML Documents Using Object-Relational Databases. *Proceedings of the International Conference on Database and Expert Systems Applications*, 206-217.

W3 Consortium. (Oct 2000). *Extensible Markup Language (XML) 1.0 (2nd Edition), Recommendation*. Retrieved January 10, 2002 from http://www.w3.org/TR/REC-xml

W3 Consortium. (May 2001). *XML Schema Part 0: Primer, Recommendation*. Retrieved January 10, 2002 from http://www.w3.org/TR/xmlschema-0

W3 Consortium. (Jun 2001). *XQuery 1.0 Formal Semantics, Working Draft*. Retrieved January 10, 2002 from http://www.w3.org/TR/query-semantics

W3 Consortium. (Nov 2001). *Semantic Web, Activity Statement*. Retrieved January 10, 2002 from http://www.w3.org/2001/sw/Activity

W3 Consortium. (Dec 2001a). *XQuery 1.0: An XML Query Language, Working Draft.* Retrieved January 10, 2002 from http://www.w3.org/TR/xquery

W3 Consortium. (Dec 2001b). *XQuery 1.0 and XPath 2.0 Data Model, Working Draft*. Retrieved January 10, 2002 from http://www.w3.org/TR/query-datamodel

W3 Consortium. (Dec 2001c). *XML Query Use Cases, Working Draft*. Retrieved January 10, 2002 from http://www.w3.org/TR/xmlquery-use-cases

W3 Consortium. (Dec 2001d). *XML Path Language (XPath) 2.0, Working Draft*. Retrieved January 10, 2002 from http://www.w3.org/TR/xpath20/

*X-DEVICE Web site*. (n.d.). Retrieved January 10, 2002 from http://www.csd.auth.gr/~lpis/systems/x-device.html

Yeh, C.-L. (2000). A Logic Programming Approach to Supporting the Entries of XML Documents in an Object Database. *Proceedings of the International Symposium on Practical Aspects of Declarative Languages (PADL)*, 278-292.