Object-Oriented Mediator Queries to XML Data

Hui Lin, Tore Risch, Timour Katchaounov
Hui.Lin, Tore.Risch, Timour.Katchaounov@dis.uu.se
Uppsala DataBase Laboratory, Uppsala University, Sweden

Abstract

The mediator/wrapper approach is used to integrate data from different databases and other data sources by introducing a middleware virtual database that provides high level abstractions of the integrated data. A framework is presented for querying XML data through such an Object-Oriented (OO) mediator system using an OO query language. The mediator architecture provides the possibility to specify OO queries and views over combinations of data from XML documents, relational databases, and other data sources. In this way interoperability of XML documents and other data sources is provided. The mediator provides OO views of the XML data by inferring the schema of imported XML data from the DTD of the XML documents, if available, using a set of translation rules. A strategy is used for minimizing the number of types (classes) generated in order to simplify the querying. If XML documents not having DTDs are read, or if the DTD is incomplete, the system incrementally infers the OO schema from the XML structure while reading XML data. This requires that the mediator database is capable of dynamically extending and modifying the OO schema. The paper overviews the architecture of the system and describes incremental rules for translating XML documents to OO database structures.

1 Introduction

The capability of database storage and processing is central in most information systems. Earlier, organizations used monolithic database management systems. However, nowadays there are often many isolated data repositories distributed over personal computers and networks of computers. Those data repositories are often heterogeneous because of the differences in the semantics of data and DBMS differences such as different data models and query languages. For those heterogeneous databases, there emerge needs to incorporate and provide the user with a unified information view.

The wrapper-mediator approach [18] has been proposed to help to solve the data integration problem. In the wrapper/mediator architecture, a *mediator*, which is an intermediate virtual database, is established between the data sources and the application using them. A *wrapper* is an interface to a data source that translates data into a common data model (CDM) used by the mediator. The user accesses the data sources through one or several mediator systems that present high-level abstractions (views) of combinations of source data. The user does not know where the data comes from but is able to retrieve and update the data by using a common mediator query language.

In this project, we investigate a method to combine and query XML documents (files and data streams) from the web through an object-oriented database mediator system. We have implemented a system that translates XML data to an OO common data model that can be queried using an OO query language similar to the object extensions of SQL-99. When available, the DTD meta-data descriptions of the XML documents are used to infer the OO schema. If there is no DTD specified the system will incrementally infer the OO schema while reading the XML documents using a set of translation rules. A straightforward translation, as is done in [3] for an extended OQL, will generate many types (classes) and make the queries clumsy, with many levels of function calls. The system therefore has a strategy to translate XML elements to functions (attributes or methods) rather than types when possible. It may then happen that the system first defines an element type as a function and then later discovers that the function must be migrated to a type to represent all uses of the element. Therefore the schema is dynamically modified during XML data reading when the object structure first inferred may not be general enough to represent the XML data read later. For this reason, incremental rules are defined that modify the schema dynamically when reading XML documents without DTDs, or when the DTD is not completely describing the XML data.

The dynamic creation of the OO schema furthermore has the effect of helping the user understand the database by looking at a generated OO schema discovered by the system.

The rest of this report is arranged as follows: Section 2 presents XML and the background to understand this paper. Section 3 describes the architecture of our object-oriented mediator system with regard to XML data sources and relates it to similar approaches. Section 4 contains the translation rules used to parse XML documents. Section 5 concludes the work and discusses possible future work.

2 Background

XML, Extensible Markup Language [5] was created as a data exchange and representation standard. XML provides ways to store complex data structures in a way suitable for exchange over the Internet. An XML document can be a file or a data stream containing nested elements starting with a root element. It may have meta-data descriptions through DTDs (Document Type Definitions). These meta-data descriptions provide some structure and constraints on the XML documents using the DTD. However XML documents may be described without DTDs and DTDs may also leave parts of the data unspecified; i.e. compared to the relational and OO data models, the XML data model is *semi-structured*.

Compared to an OO data model the XML data model does not have classes, methods, and inheritance; instead it has *element types* and *attributes* [5] which are similar to classes and attributes in OO data models. Thus XML does not use a complete OO data model. In order to avoid confusion in the discussion below we use the term *element tag* (or just *tag*) to mean element type.

Example 1 shows a small DTD named *person.dtd*. In this example, the DTD restricts the tag *person* to always contain a sub-element tagged *employee*, and each element tagged *person* always has the attribute *id*. We call an *!ELEMENT* statement *element definition*. Elements tagged *employee* always have two sub-elements tagged *family* and *given* both of which have the system attribute "*#PCDATA*". "*#PCDATA*" indicates that the element may have a text string as its value. The element definitions specify containment relationships to sub-elements along with constraints on the order and occurrences of sub-elements. In the example the sub-element tagged *given* always must follow *family* inside an element tagged *employee*. The element tagged *email* can have any other element as sub-element (indicated with *ANY*); we say that the sub-elements of *email* are *unspecified*.

```
<!ELEMENT person (employee)>
<!ELEMENT employee (family, given)>
<!ELEMENT family (#PCDATA)>
<!ELEMENT given (#PCDATA)>
<!ELEMENT email ANY>
<!ATTLIST person id ID #REQUIRED>
```

Example 1 The DTD *person.dtd*

```
<!DOCTYPE person SYSTEM "person.dtd">
<person id = "669">
    <employee>
        <family> Lin </family>
        <given> Hui </given>
    </employee>
</person>
```

Example 2 An XML document

Example 2 is an XML document using the DTD *person.dtd*. The element tagged *person* has an attribute *id* whose value is "*669*" and a subelement tagged *employee* that contains two subelements: the element tagged *family* whose value is "*Lin*" and the element tagged *given* whose value is "*Hui*". Every XML document must have a *root element* specified in the header; in this case it is the element tagged *person*.

Several query languages have been proposed for XML documents [15]. A graphical query language is introduced in [2] where queries over XML documents are specified graphically. A pattern-based query language is proposed in [20] where regular path expressions are used to match XML structure and data, and derive new XML data.

Lore [9] is a database management system for storing and querying XML documents. Lore maps a XML document into a semi-structured directed, labeled, and ordered graph data model called OEM [14]. In [8][17] methods are developed to store XML data in relational databases.

3 Object-oriented data mediation over XML data sources

The purpose of our work is to transparently query XML documents from an OO mediator. By wrapping an XML data source the user does not need to know from where the XML document originates and will be able to combine many XML data sources and query them by using an OO query language.

In our approach we create an OO schema by translating the DTD according to some translation rules. When no DTD is present, or when the DTD is incomplete, the OO schema is incrementally created while the XML document is read. Thus a structured OO schema is incrementally discovered from DTDs or from XML data. Such a schema provides semantically enriched meta-data to guide the user when querying the database. It also provides a basis for data indexing and efficient query processing [1].

The XML data is translated into objects when read into the OO mediator database. In order to query such data with an OO query language the following facilities are needed:

- an OO storage manager to represent object OIDs;
- an OO query processing system;
- a convention for what constitutes an OO database schema from a set of XML documents;
- a translation mechanism from XML data to objects in the mediator.

We use an OO data model to which XML data is translated similar to what is proposed for SGML in [3]. In [12] a strategy is devised to generate a graph called the *Data Guide* that summarizes containment relationships among XML elements using the OEM graph data model. In our case we generate the containment relationships as both type (class) and function (attribute) definitions. Unlike [3] we use a strategy to avoid creating types in order to simplify the schema and subsequent querying.

In our system all XML documents having the same DTD are regarded as one data source having an OO schema inferred from the DTD. The schema of the mediator combines the imported schemas, and the mediator database view covers the union of the accessed XML documents. We do not require every XML document to have a DTD as in [3] but incrementally generate and modify the schema while reading XML documents when no DTD is available or the DTD is incomplete. Our system can thus handle XML documents with DTDs, with incomplete DTDs, or without DTDs.

When DTDs are translated to OO schemas, some semantic enrichment is made to infer types and attributes from the DTDs in order to simplify the schema according to the rules below. Whenever a new XML document having a DTD is accessed from the mediator we check what DTD it refers to, if any, and whether this DTD previously has been translated to an OO schema in the mediator. If the DTD is not translated beforehand the mediator will read the DTD to infer types (classes) and functions (attributes). XML documents having no DTD at all are regarded as belonging to a special schema.

We have implemented our approach using the object-oriented, lightweight, and extensible database mediator system AMOS II [16]. It provides a main memory data manager to store materialized XML data, and a query processing engine. AMOS II has an object-oriented query language, AMOSQL, similar to OSQL [7] and the OO extensions of SQL-99.

In addition to basic OO data management facilities AMOS II provides facilities for data integration by combining data from distributed and heterogeneous data sources using the mediator/wrapper approach. By utilizing data mediation facilities of AMOS II [6][10][11] we can query combinations of XML data, relational databases and other kinds of data sources, as illustrated in Figure 1. It shows an example of distributed mediation with AMOS II where three applications access data from four heterogeneous data sources through three distributed mediators.

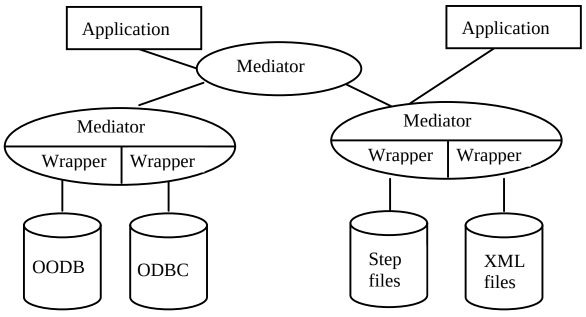

Figure 1 An example of distributed AMOS II Systems

The AMOS II data model contains three basic constructs: *objects*, *types* and *functions*. *Objects* are used to model all entities in the database. *Types* (i.e. classes) are used to describe the object structure and they are organized in an OO type hierarchy of subtypes and supertypes. *Functions* are defined on types and are used to represent properties of objects and relationships between objects. Functions thus represent attributes and methods. The AMOS II system is extensible through several interfaces to its kernel – C, Java, and Lisp, respectively. In this work, we utilize the Java interface to build an XML wrapper for AMOS II. Our implementation materializes read data in the AMOS II storage manager and we have developed a translation mechanism from XML to the AMOS II OO data model.

The translation mechanism includes:

- A strategy for generating schemas from DTDs when available. The OO schema is derived from the DTD in this case and it describes the contents of one or several XML documents referencing the same DTD.
- A strategy for incrementally populating the database using the generated OO schemas while reading XML data. Thus OO database update statements are called while XML data is read.
- Strategies for dynamically extending the schema when reading XML data with no DTD or when the DTDs are not fully describing all the data. In this case both database update and schema modification statements are dynamically called while XML data is read.

The AmosXML architecture in Figure 2 is based on the existing system in a non-intrusive way, requiring no modifications to AMOS II. It uses a Java interface to interact with the kernel of the system and is implemented as a set of foreign functions implemented in Java. Those functions are called from AMOS II to load DTD schema and XML data into the database. The read data can be queried by the existing query capabilities of the AMOS II system.

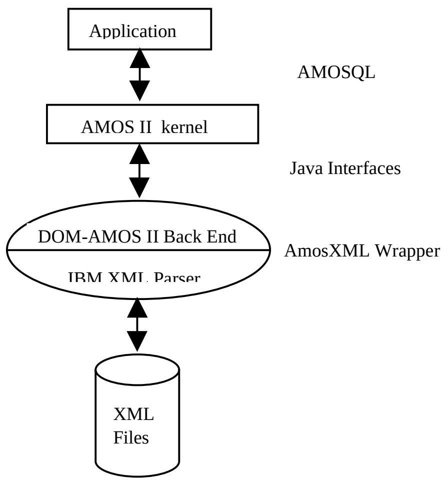

Figure 2 AmosXML Wrapper

In the AmosXML wrapper, IBM's XML Parser for Java [19] is used to parse XML documents into the Document Object Model (DOM) data structure [4]. The DOM is a platform and language neutral interface that allows programs and scripts to dynamically access and update the contents, structures and styles of documents. The DOM provides a standard set of objects for representing HTML and XML documents, a standard model of how these objects can be combined, and a standard interface for accessing and manipulating them. It closely resembles the structure of the documents it models. Using DOM as input, the "DOM-AMOS II Back End" communicates with AMOS II to call AMOSQL statements dynamically. The user finally submits AMOSQL queries to the AMOS II server to access the XML data.

4 Translation rules

One challenging issue is to define a way to map the data model of an XML document into the data model of AMOS II. The DTDs produce schema information and the XML documents themselves produce data but also schema information when DTDs are omitted or incomplete. Therefore we design the schema definition rules to be incremental; i.e. the schema is dynamically extended while the XML data is read and may be modified to satisfy new data constraints discovered during this.

There are two kinds of transformation rules:

- Rules applied on DTDs;
- Rules applied while reading XML data.

In the translation rules below we will show how different kinds of DTD definitions and elements in XML documents dynamically extend the schema and content of the mediator database. We use the syntax of AMOSQL to illustrate this.

4.1 DTD rules

The first rule creates a type (class) for an element definition:

Rule 1. A new type is created for every element definition declared in the DTD to have at least one sub-element or attribute or where a subelement is declared as "ANY" or "EMPTY".

For example, assume we have the following DTD statement:

<!ELEMENT person (employee)>

Using the above rule, the following statement instructs AMOS II to dynamically extend the schema with a new type named *person* being a subtype of the system type *xml*:

create type person under xml;

The AmosXML wrapper dynamically executes the statement using the Java interfaces. Since the DTD specifies that elements tagged person must always contain one sub-element tagged employee, a new database type named person is created. Objects of that type are created when reading elements from the XML file, according to the rules in the next subsection. The system type "*xml*" is a supertype of all XML types. It has a function *data* to store the values of elements when "*#PCDATA*" is specified.

Rule 2 creates a function, rather than a type, for each leaf element defined in the DTD. By creating such *property functions* the queries to the database become simpler containing fewer levels of indirection, as shown below.

Rule 2. If an element definition E does not have any subelement definitions (i.e. is a leaf element), then tag E is represented as a stored function (i.e. attribute), also named E, on the type (class) representing its parent element definition F. The function represents the value of an element as a string, i.e. its signature is E(F)->charstring. This function is called a property function.

Consider the following part of the DTD *person.dtd* above:

<!ELEMENT employee (family, given)>
<!ELEMENT family (#PCDATA)>
<!ELEMENT given (#PCDATA)>

After applying Rule 2 in our example, the following AMOSQL statements dynamically defines two new functions in the mediator:

create function family(employee) ->charstring as stored;
create function given(employee) ->charstring as stored;

The clause "*as stored*" specifies that the functions represent attributes (i.e. contain explicitly stored values).

In this case the element definitions *family* and *given* do not contain any subelements, so two property functions are created returning the type *charstring*, rather than two new types. An example of a query to this database is:

select family(e) from employee e where given (e)= "Hui";

If we would not have Rule 2, *family* and *given* would have been represented as types (classes) with the values of the elements stored in the function *data*. We would then have the following schema created instead:

create type family under xml;
create type given under xml;

The above query would then have looked like this:

select data (f) from family f, given g, employee e
where data (given(e)) = "Hui";

This is clearly a more complex and less natural query. By representing leaf elements as functions most calls to the *data* function are avoided and fewer types (classes) are needed. With Rule 2 the *data* function needs only be used for accessing elements having both subelements and *#PCDATA* specified.

For elements definitions represented in the mediator as types having subelements also represented as types, Rule 1 generates a new type. The following rule then generates a *containment function* to represent the element-subelement relationship between the new type and the elements it contains:

Rule 3. A <u>containment function</u> is generated for an element definition represented in the mediator as a type (class) F that has a subelement E also represented as a type (class). It returns the collection of subelement objects contained in a given object. Its signature is E(F)->bag of E.

For example, consider the following DTD:

<!ELEMENT person (employee)>
<!ELEMENT employee (family, given)>

It generates the following containment function definition:

create function employee(person) ->bag of employee
as stored;

The containment function *employee(person)->bag of employee* returns a bag (set with duplicates) of employees since the element definition *employee* has one or several subelements.

Attribute definitions generate functions prefixed with "*attribute_*" to distinguish them from property functions:

Rule 4. An <u>attribute function</u> is created for each XML attribute defined in a DTD (using ATTLIST) to represent the attribute of each element.

For example,

<!ATTLIST person id ID #REQUIRED>

is translated into the following function definition:

create function attribute_ id(person) -> charstring
as stored;

A function *attribute_id(person)* represents values of the attribute *id* for objects of type *person.*

4.2 XML data rules

The XML data rules dynamically add data to the mediator database while reading XML documents. They may also modify the schema in case the DTD is incomplete or omitted.

Rule 5 concerns creating objects for elements represented as types:

Rule 5. When reading an element, a test is made to check if its tag is previously represented as a type. If so, a new object O of that type is created. If the element has a value the data function data(O) is set to the contents of the element. If O is not the root element it is also added to the containment function of its parent element.

For example, if the tags *person* and *employee* are previously represented as types, the following XML document

```
<!DOCTYPE  person SYSTEM  person.dtd>
<person>
        <employee>
             Lin
        </employee>
</person>
```

generates calls to the following AMOSQL statements:

create person instances :p;
create employee instances :e;
set data(:e) = "Lin";
add employee (:p) = :e;

In this example, Rule 5 first generates a new object of type *person* using the statement *create.* It has no value and is the root element of the document. Then the same rule creates an object of type *employee* and the *data* function of the new object is set to the contents of the element using the *set* statement. The new object is also added to the extent of the parent's containment function *employee(person)->bag of employees* using the *add* statement.

Rule 6 complements Rule 5 when the tag was previously represented as a property function rather than as a type:

Rule 6. If the tag of an element is previously represented as a property function on its parent (rather than as a type), it is added to the extent of that property function.

For example, if the tag *person* is previously represented as a type, and tag *email* is not previously represented as a type, but as a property function *email(person)->charstring,* the following XML document

<!DOCTYPE person SYSTEM "person.dtd">
<person>
<email>
Hui.Lin@dis.uu.se
</email>
</person>

generates calls to the following AMOSQL statements:

create person instances :p;
add employee(:p) = "Hui.Lin@dis.uu.se";

In this example, Rule 5 generates a new object of type *person* using the statement *create.* It has no value and is the root element of the document. Then we populate the containment function:

employee(person)->charstring.

The next rules apply when there is no DTD or when the DTD is incomplete ("*ANY*" specified). In such cases the schema is either extended or modified depending on how the tag is represented so far.

The following definition is used below:

***Definition**: A sub-element E of F is an unspecified subelement of F if either no DTD definition exists for E or F is specified as "ANY" in the DTD.*

Rule 7 dynamically creates a new containment function the first time a containment relationship is discovered while reading an XML document:

Rule 7. If 1) element tagged E is an unspecified subelement of an element tagged F; 2) E is represented as a type; 3) there is no previously defined containment function E->F; then we dynamically create a new containment function E->F.

Assume now the following XML document without DTD:

<person>
<employee>
<family> Lin </family>
</employee>
</person>

In this case Rule 7 applies since

- *employee* is an unspecified subelement of person (it is the first time that employee shows up) ,
- tag employee is represented as a type (Rule 1),
- there is not previous containment function *employee(person) -> bag of employee.*

Following Rule 7, the following containment function is dynamically generated when the XML document is read:

create function employee(person) ->employee as stored;

The containment function is created in order to establish the discovered containment relationship between types *person* and *employee*.

The next rule complements Rule 7 when E is not a type:

Rule 8. If 1) element tagged E is an unspecified subelement of an element tagged F; 2) E is not previously encountered; 3) the element tagged E has no subelements; then we dynamically create a property function E(F)->charstring.

Assume the following XML document not having any DTD:

<person>
<email> Hui.Lin@dis.uu.se </email>
</person>

In this case Rule 8 applies since the element *email* is an unspecified subelement of the element *person*, tag *person* is not previously defined as a type, and the element has no subelement. The following function definition is dynamically created:

create function email (person) ->charstring as stored;

Notice that this rule will apply only once for each kind of element.

Rule 9 dynamically converts a previously defined property function to a type when the element is discovered to have subelements:

Rule 9. If 1) an element tagged E is an unspecified subelement of an element tagged F; 2) E is previously represented as a property function E(F)->charstring; 3) the element tagged E has its own subelements or attributes; then we dynamically create a new type representing E and a containment function with signature E(F)->bag of E. The property function E(F)->charstring is converted into the containment function and the containment function is updated accordingly.

For example, assume the following XML document without associated DTD:

```
<person>
    <name>Hui Lin </name>
</person>
<person>
    <name><family> Lin </family></name>
</person>
```

According to Rule 8, the following statements are generated when parsing the first element *person*:

```
create type person under xml;
create function name (person) -> charstring as stored;
create person instances :p;
set name (:p) = "Hui Lin";
```

A new type is created when reading the first occurrence of element *person* since it contains subelement *name*. Since element *name* has no subelement a property function *name(person)->charstring* is created and populated.

However, when later reading the second occurrence of element *person*, an element *family* is discovered as a subelement of the element *name*. Thus the property function *name* must be migrated to a type. According to Rule 9, the following statements are generated:

```
create type name under xml;
create name instances :m;
set data(:m) = name(:p);
create function name (person) -> name as stored;
set name(:p) = :m;
create person instances :pn;
create name instances :n;
set name(:pn) = :n;
```

In this case, a new type is created for the element *name* since it contains a subelement *family*. The property function *name(person)->charstring* is converted into a containment function *name(person)->name*, and populated with *:m* (from the old property function) and *:n* (the new object).

Furthermore, we also need to generate a property function *family(name)->charstring* according to Rule 8, since the element *family* has no subelement.

```
create function family(name)->charstring as stored;
set family(:n) = "Lin";
```

Finally we need a special overriding rule to guarantee that there will always be an object representing root elements and elements having subelements or attributes:

Rule 10. Tags of elements that are found to 1) be root element; 2) or have sub-elements; 3) or have attributes; are always represented as types.

For example, in the uncommon event that the root element of an XML document does not have any subelements it will be represented as an object. We will not elaborate this case further here.

5 Conclusion and future work

We described the architecture of a wrapper called AmosXML that allows parsing and querying XML documents from an object-oriented mediator system. Furthermore, incremental translation rules were described that infer OO schema elements while reading DTD definitions or XML documents. Some rules infer the OO schema from the DTD, when available. For XML documents without DTDs, or when the DTD is incomplete, other rules incrementally infer the OO schema from the contents of the accessed XML documents. The discovery of OO schema structures combined with other OO mediation facilities in AMOS II [6][10][11] allow the specification OO queries and views over data from XML documents combined with data from other data sources. The incremental nature of the translation rules allow them to be applied in a streamed fashion, which is important for large data files and when the network communication is slow or bursty.

There are several possible directions for our future work:

- The current rules do not infer any inheritance, but a flat type hierarchy is generated. Rules should be added to infer inheritance hierarchies, e.g. by using behavioral definitions of types [13] where a type is defined by its behavior (i.e. its attributes and methods). In our case this means that a type T is defined as a subtype of another type U if the set of functions on U is a subset of the set of functions on T.
- Integrating XML data involves efficient processing of queries over many relatively small XML data files described by several layers of meta-data descriptions and links. For example, there can be 'master' XML documents having links to other XML documents and DTDs. Therefore the query language needs to be able to transparently express queries referencing both XML data and the meta-data in the master documents. New techniques are needed to be able to specify and efficiently process queries over such multi-layered meta-data.
- The conventional exhaustive cost-based query processing techniques do not scale over large numbers of distributed XML documents. New distributed heuristic techniques need to be developed for this.

References

[1] K.Böhm, K.Aberer, M.T.Özsu, K.Gayer: Query Optimization for Structured Documents Based on Knowledge on the Document Type Definition, *Proc. IEEE ADL'98 Conf.*, Santa Barbara, CA, 1998.

[2] S. Ceri, S. Comai, E. Damiani, P. Fraternali, S. Paraboschi, L. Tanca: XML-GL: A Graphical Language for Querying

and Reshaping XML Documents, *http://www.w3.org/TandS/QL/QL98/pp/xml-gl.html.*

[3] V.Christophides, S.Abitteaboul, S.Cluet and M.Scholl: From Structured Documents to Novel Query Facilities, *ACM SIGMOD Conf.*, Minneapolis, 1994.

[4] Document Object Model (DOM) Level 1 Specification, Version 1.0, *http://www.w3.org/TR/PR-DOM-Level-1.*

[5] Extensible Markup Language (XML) 1.0, *http://www.w3.org/TR/1998/REC-xml-19980210.*

[6] G. Fahl, T. Risch: Query Processing over Object Views of Relational Data, *VLDB Journal* , Vol. 6 No. 4, November 1997, pp 261-281.

[7] D.Fishman, D.Beech, J.Annevelink, E.Chow, T.Connors, J.Davis, W.Hasan, C.Hoch, W.Kent, S.Leichner, P.Lyngbaek, B.Mahbod, M-A. Neimat, T.Risch, M-C Shan, W.Wilkinson: Overview of the Iris DBMS, in W. Kim, F. H. Lochovshy (eds.), *Research Foundations in OO and Semantic,* Addison-Wesley, pp. 174-199, 1990.

[8] D.Florescu, D.Kossmann: A Performance Evaluation of Alternative Mapping Schemes for Storing XML Data in a Relational Database, INRIA Technical Report, INRIA, No. 3680, May, 1999.

[9] R.Goldman, J.McHugh, and J.Widom: From Semisturctured Data to XML: Migrating the Lore Data Model and Query Language, *WebDB* (Informal Proceedings) 1999.

[10] V.Josifovski, T.Risch: Functional Query Optimization over Object-Oriented Views for Data Integration, *Intelligent Information Systems (JIIS),* Vol. 12, No. 2/3, Kluwer, 1999.

[11] V.Josifovski, T.Risch: Integrating Heterogeneous Overlapping Databases through Object-Oriented Transformations, *25th VLDB Conf.,* Edinburgh, Scotland, Sept. 1999.

[12] S.Nestorov, J.Ullman, J.Wiener, S.Chawathe: Representative Objects: Concise Representations of Semistructured, Hierarchical Data, *IEEE ICDE Conf.,* Birmingham, UK, April 1997.

[13] M.T.Özsu, R.Peters, D.Szafron, B.Irani, A.libka, A.Muñoz: TIGUKAT: A Uniform Behavioral Objectbase Management System, *VLDB Journal,* Vol. 4, No. 3, July 1995.

[14] Y.Papakonstantinou, H.Garcia-Molina, J.Widom: Object Exchange Across Heterogeneous Information Sources, *IEEE ICDE Conf.,* Taipei, Taiwan, March 1995.

[15] QL'98 - Position Papers, *http://www.w3.org/TandS/QL/QL98/pp.html.*

[16] T,Risch, V.Josifovski, T.Katchaounov: AMOS II Concepts, *http://www.dis.uu.se/~udbl/amos/doc/amos_concepts.html*

[17] J.Shanmugasundaram, K.Tufte, G.He, C.Zhang, D.DeWitt, J.Naughton: Relational Databases for Querying XML Documents: Limitations and Opportunities, *25th VLDB Conf.,* Edinburgh, Scotland, 1999.

[18] G.Wiederhold: Mediators in the Architecture of Future Information Systems, *IEEE Computer*, Vol. 25, No. 3, March 1992.

[19] XML Parser for Java, *http://www.alphaworks.ibm.com/formula/xml*.

[20] XML-QL: A Query Language for XML, *http://www.w3.org/TR/1998/NOTE-xml-ql-19980819.*