Integrating Relational Databases and XML Technology: the ERX Tool

Giuseppe Psaila, Davide Brugali

Facoltà di Ingegneria, Università degli Studi di Bergamo,
V.le Marconi 5, 24044 Dalmine, Italy
{psaila, brugali}@unibg.it

Abstract. This paper reports about our experience in developing the ERX Data Management System, a system devised to collect data coming from different XML data sources, and store them into a database in a way independent of the source format; its query language, named ERX-QL, is able to query the database and generate new XML documents. We developed the ERX Data Management System to explore the possibility of integrating three different basic technologies, Relational DBMS, Java and XSLT, under a unifying framework which makes the system interoperable w.r.t. the particular adopted technology (for example Relational vs Object-Oriented database technology); hence, this framework is based on an Entity-Relationship-like Data Model (ERX), which is not tied to any specific technical and/or commercial solution. The paper discusses the architecture of the ERX system, and the adopted technical solutions.

1 Introduction

B2B applications usually deal with a large variety of different documents, belonging to several document classes. These documents are usually strongly correlated, because a single document is focused on a limited view of the overall business process; hence, an information system has to gather and integrate documents, in order to avoid redundancy and ensure correctness.

XML is becoming the standard format to exchange information over the internet. Its characteristics make XML suitable for a variety of applications, in particular those that exchange documents and information with heterogeneous data sources and information systems.

Although the W3C Recommendation that introduced XML [1] allows a generic XML processor to be able to process a generic XML document, when XML documents are automatically processed by information systems, exchanged XML documents cannot be generic; instead, they must belong to well precise document classes, where mark-ups and their semantics are accepted by all communicating actors.

The previous considerations motivated our work. We developed a system able to store data coming from XML documents, with a suitable query language to retrieve data and generate new XML documents. This system is called the ERX Data Man-

agement System. It is devoted to deal with XML data, but we decided not to provide a data model based on the tree structure of XML documents. In contrast we wished a data model able to clearly describe concepts in the data. Our choices are motivated by the following perspective use of our system. XML documents belonging to different classes come to the system; the goal is to store relevant data they describe, in a form independent of the XML structure. This way, the data model can be independent of any particular DTD or XML Schema definition, and can be focused on relevant concepts in the data.

Data are successively retrieved and assembled to produce new XML documents, possibly obtained by aggregating information received through several XML documents, even belonging to different classes. This can be achieved only by means of a suitable query language, that extracts the desired information and generates XML documents.

We decided to experiment a system architecture that integrates a relational database, Java components and XSL Style-sheets, in order to evaluate the feasibility of such a solution; in particular, the relational database store data, java components provide procedural functions to access the data, XSL Style-sheets are responsible for actually dealing with XML documents, in order to make java components unaware of the real structure of XML documents.

As far as the choice of the data model is concerned, the relational data model provided by the relational database is not satisfactory, due to the fact that the relational model is not able to capture concepts naturally described by the tree structure of XML documents. For example, concept hierarchies, that come out from XML documents, are not naturally captured by the relational model. Hence, a higher level data model was considered suitable for our system, while a RDBMS is used for data storage.

The data model is called ERX (Entity-Relationship for XML) and is an extended Entity-Relationship data model, specifically designed to cope with specific features of data coming from XML documents. The advantages are several: ERX is independent of the data model provided by the database; an ERX schema clearly describes concepts and complex relationships between concepts described in XML documents (better than a relational schema); the ERX schema is not tailored on a specific DTD or XML Schema specification, but can cover data coming from documents possibly belonging to several classes of documents; the ERX schema can be easily queried in order to perform complex selection or aggregation of data, and generate new XML documents resulting by the aggregation of data coming from several documents.

In this paper, we describe the ERX Data Management System, discussing in detail the adopted technical solutions, the role of each component and the results obtained in the integration of three different technologies. We will see that the reached degree of integration is significant, even though the degree of interoperability among components remains high.

The paper is organized as follows. Section 2 informally introduces the ERX Data Model . Section 3 introduces the ERX Query Language. Section 4 discusses the

architecture of the system, and the role of components. Section 5 discusses the implementation. Finally, Section 6 draws the conclusions.

2 The ERX Data Model

Before we start introducing the ERX Data Model, it is necessary to stress the philosophy behind this data model and the system. The ERX Data Management System aims to be a *data management system*, which provides functionality to *manage data*; in particular, this functionality are tailored on the fact that data come from XML documents and that the output must be XML documents. Hence, the data model cannot be XML or a model that maintains the tree structure (as Lorel [2] or Tamino [3]. Furthermore, we wanted a data model independent of the particular database technology, but always able to clearly describe concepts in the data. Hence, the system is able to load XML documents into the ERX database and generate documents from data in the ERX database, but is not based on the same syntactic structure of XML.

2.1 Modules

An ERX Schema is a collection of *modules*. A module groups together homogeneous concepts which are strongly correlated. A module can be seen as a container of entities and relationships. Graphically, a module is represented as a rectangle containing entities and relationships.

Modules introduce a modularization mechanism: modules can be defined in isolation and then assembled together by means of a form of relationship, called Inter-Module Relationship (discussed later).

This concept becomes very useful when data coming from several XML document classes are stored into the ERX database: in fact, strongly correlated concepts can be grouped together; a module might correspond to a DTD or a portion of DTD; furthermore, it might describe concepts which are present in different DTDs.

2.2 Entities

Inside a module, an entity describes a complex (structured) concept of the source XML documents. Entities are represented as solid line rectangles; the entity name is inside the rectangle.

An instance of an entity X is a particular occurrence of the concept described by entity X in a source document. It is identified by a unique, system generated, numerical property named OID.

2.3 Relationships

A *relationship* describes correlations existing between entities X and Y. A relationship is represented as a diamond labeled with the name of the relationship. The diamond is connected to X and Y by solid lines; these lines are labeled with a *cardinality constraint* (l :u), which specifies for each instance of entity X (resp. Y) the minimum number l and the maximum number u of associated instances of Y (resp. X). An *instance* of the relationship describes a particular association between two instances of the connected entities.

A complex form of relationship is represented by a relationship with alternatives: an instance of an entity X is associated with instances of alternative entities Y_1, Y_2, ..., Y_n; the cardinality constraint for X considers all associations of an instance of X with instances of any entity among Y_1, Y_2, ..., Y_n.

Orthogonally, a relationship can be a containment relationship. Given two entities X and Y, a containment relationship from X to Y denotes that an instance of X structurally contains instances of Y. Containment relationships are good only for those situations in which associations are ordered, for example ordered lists. Considering XML documents, containment relationships are then good for describing semi-structured or mixed tag content.

2.4 Attributes

Entities can have attributes: they represent elementary concepts associated to an entity. Attributes are represented as small circles, labeled with the name, and connected to the entity they belong to by a solid line. Entity attributes are always string valued. Furthermore, ERX does not provide the concept of key attribute. Considering XML documents, attributes can be used to represent both attributes appearing in XML tags and textual content of tags; the fact that XML does not consider types for attributes, motivates the choice for having only string valued ERX attributes. The fact that the ERX Data Model does not consider key attributes is motivated by the absence of an analogous concept in XML. In fact, XML attributes defined as ID are used in XML documents only to realize cross-references.

2.5 Hierarchies

Specialization hierarchies are possible in ERX. If an entity X is specialized into several sub-entities Y_1, Y_2, ..., Y_n, this means that an instance of the super-entity X is actually an instance of one of the sub-entities Y_1, Y_2, ..., Y_n. All the attributes of X are common to (or inherited from) all the sub-entities, which can have specific attributes. ERX hierarchies are only total and exclusive: an instance of the root belongs to one and only one leaf entity.

2.6 Interfaces and Links

The last concept considered inside ERX modules is the concept of interface. Through an interface, it is possible to link entities to other modules, and at the same time they define which are the main concepts for modules. Interfaces are represented as dashed line rectangles adjacent to the module border. Entities are connected to interfaces be means of links. A link is represented by a labeled circle with solid lines, one from the circle to the interface, one or more from the circle to the entities. Multiple entities connected to the same link are alternatives, i.e. from the interface it is possible to reach instances of several entities. An interface can be connected to one link only.

2.7 Inter-Module Relationships

Modules represent homogeneous concepts that can be found in XML documents. However, it may be necessary to create correlations between modules. This is possible by means of inter-module relationships. Inter-module relationships are similar to intra-module relationships: they connect module interfaces instead of entities; they can have alternatives, and can be containment relationships.

```
<Products>
  <Brand>
   <Name> SC </Name>
   <Address> New York,USAy </Address>
   <URL> http://www.sc.com </URL>
   <Email> helpdesk@sc.com </Email>

   <Durable PID="C001">
      <Description>
        Computer 500 MHz, RAM 128MB, Hard disk  10GB
      </Description>
      <Technical>
         Technical info can be found at
         <Hyperlink URL=http://www.sc.com/G001.html
               Text="C001 Product page"/>
      </Technical>
      <Note> This PC supports Windows 2000</Note>
   </Durable >

   <Consumable PID="FD001">
     <Description> Floppy Disk </Description>
      <Technical> High density floppy disk</Technical>
      <UsedBy PID="C001"/>
       <UsedBY PID "C002"/>
   </Consumable>
  </Brand>
</Products>
```

Fig. 1. An example of XML file describing a product

2.8 An example

Let's consider an example related to a products catalogue . Each producer provides information about brands and products. An example is the XML document reported in Figure 1, where products with brand SC are described.

The structure of the document is intuitive: each brand is characterized by name, address, URL of its web site, e-mail of its customer service. A sequence of tags **Durable** and **Consumable** describe each single product, distinguishing between durable products and consumable products. In particular, the content of tag **Product** is constituted by a short description, denoted by tag **Description**, a technical description, denoted by tag **Technical**, and a possibly missing sequence of notes, denoted by tag **Note**. Observe that tags **Technical** and **Note** are allowed to contain text mixed with hyperlinks (the empty tag Hyperlink). Tag **Consumable** can also contain an empty tag named **UsedBy**, whose attribute **PID** denotes the code of a product that uses the described consumable product. In contrast, tag **Durable** can contain nested **Durable** tags, describing the components assembled to form an assembly product.

Figure 2 reports the corresponding ERX model.

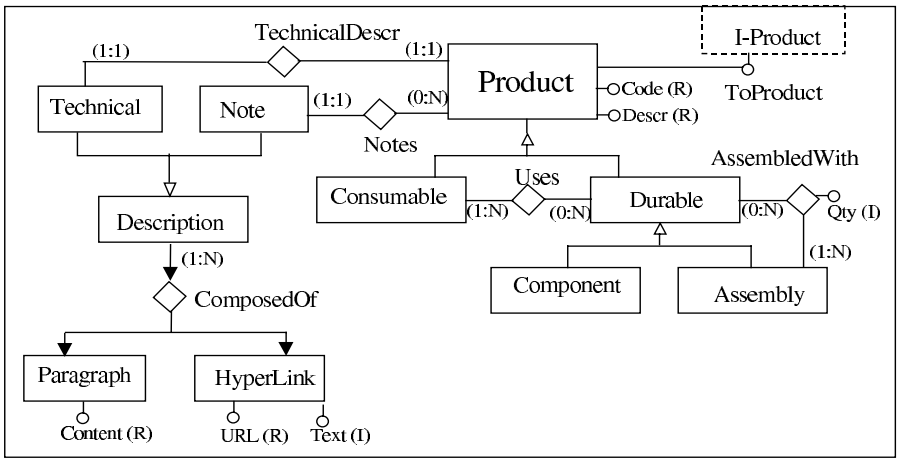


Fig. 2. An example of ERX model describing a product

3 The ERX Query Language

As previously introduced, the ERX Data Management System provides a suitable query language, to extract information from within the ERX database and compose new XML documents. The ERX Query Language (ERX-QL) is in the middle of two worlds: from one side, it operates on the ERX database to extract entity instances, by navigating the ERX Schema; from the other side, it generates XML documents as output. Consequently, the closure property typical of relational algebra does not hold for ERX-QL.

ERX.QL allows generation of nested XML structures, even recursively. This is achieved by the notion of *named query*: this is an ERX-QL query stored (either temporarily or persistently) by the system. Other queries can call previously defined named queries, in order to perform very complex tasks. Since the ERX-QL query calling mechanism allows recursion (direct of indirect), complex recursive XML structures derived by navigating circularities in the ERX Schema can be easily specified. Furthermore, ERX-QL provides the concept of *query library*. It is possible to associate to the ERX Schema a set of libraries, where each library is a collection of named queries. This solution allows to create a pool of standard or basic queries, that can be reused to formulate complex queries. In order to be coherent with the rest of the project, ERX-QL queries are themselves XML documents. We defined a specific set of tags, corresponding to ERX-QL constructs.

A complete description of ERX-QL can be found in [4].

4 System Architecture

We can now report about the architecture of the ERX Data Management System. We first provide an overview of the architecture, discussing the main choices we performed. Then, we illustrate in details the execution of queries written in ERX-QL.

2.1 Architecture Overview

The architecture of the ERX Data Management System has been designed to take into account the modularity and interoperability requirements that are the foundations of our project. In fact, we think that this is a successful way to reach the aimed integration among different technologies; in particular, we had to integrate a Relational DBMS with APIs written in Java and driven by XSL Style-sheets, as we discuss below. Figure 3 shows the architecture. This is organized in a very modular way, both horizontally and vertically. Let us start discussing the vertical modularization.

4.2 Vertical modularization

The ERX Data Management System is vertically modularized in four distinct layers.

Data Layer . The lower layer is the *Data Layer*, constituted by a Relational DBMS; specifically, we are using MS SQL Server 7.0. The role of the Relational DBMS is to actually store and retrieve data. In particular, the DBMS stores both the meta-data, constituted by the ERX Schemas, and the document data. The real relational data structure is completely hidden by the system. Notice that the presence of a relational DBMS ensures that the system is based on a stable core technology. This is important from two distinct points of view: from one side, RDBMSs provide func-

tionality for mirroring, backup and transaction support; from the other side, RDBMSs will be supported for a very long time by DBMS producers.

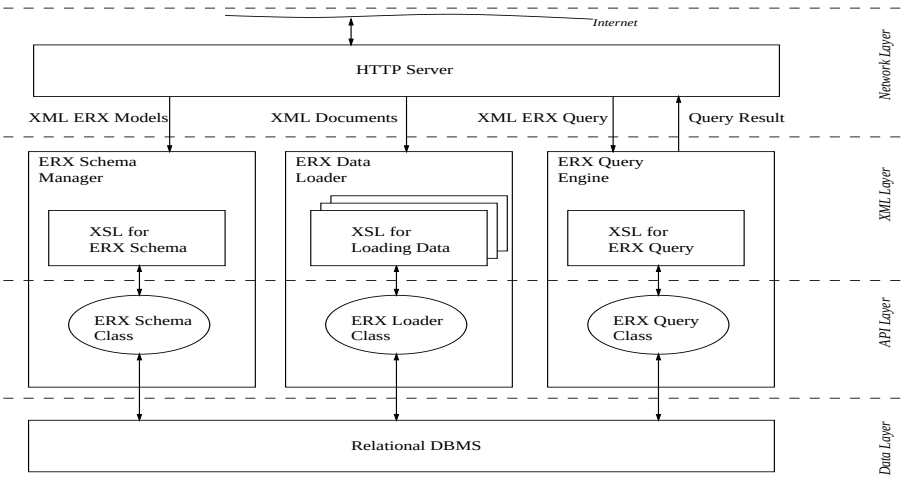


Fig. 3. The ERX system architecture

API Layer. The second layer is the *API Layer*. Java classes provides an API access method to ERX Data, that might be exploited by external applications, e.g., Java applications. The API layer provides basic primitives.

- Data Definition primitives allow the specification of an ERX schema. By means of them it is possible to define modules and inter-module relationships, to define entities, relationships, links, and so on; these primitives are also responsible to check for the correctness of the specified schema. Finally, it drives the RDBMS to create the necessary tables.
- Data Manipulation primitives allow to insert new data, by creating entity instances and relationship instances. They also allow to retrieve data, selecting entity and relationship instances and composing them to obtain the desired information.

XML Layer. The *XML Layer* is built over the API layer, and allows to interact with the system by means of XML documents. In particular, we defined a class of XML documents (and the corresponding DTD) to define ERX Schemas and we defined a class of XML documents to formulate queries also the loading of source XML documents is performed by this layer. In our system, we make use of XSL style-sheets [5] to implement this layer, which incorporates the SAXON XSL interpreter. The result is a totally modular architecture, that can be easily adapted to further developments or new standards. Furthermore, such a solution provides fast implementation: in effect, a traditional solution based on an XML parser requires a significant amount of time to spend in writing code to navigate the tree generated by the parser; in contrast, by using XSL we are decoupled w.r.t. the specific parser used in the XSL processor; furthermore, we have to write significantly less code, because the declarative style of XSL style-sheets makes the extraction of information from

within the parsed XML document very easy. The maintenance of the XML part is improved too, because it is decoupled from the procedural part.

Network Layer. Finally, the *Network Layer* makes available services provided by the ERX Data Management System to remote applications. This layer exploits a standard HTTP Server, to make possible remote access via the HTTP protocol: the system receives XML documents and sends XML documents.

Notice that the architecture of the system is interoperable w.r.t. the relational representation. In fact, the substitution of the RDBMS with a different technology affects only two of the four layers, i.e. the Data Layer and the API Layer. Hence, this architecture can be easily adapted to new successful technology solutions, because the interface provided by the API layer is the ERX Data Model, which is independent of any specific DBMS.

4.3 Horizontal Modularization.

We now discuss the horizontal modularization. This concerns the API and XML layers, that are subdivided in three distinct tools.

ERX Schema Manager. This tool receives the XML specification for the ERX Schema, checks its correctness, updates the Meta Schema DB, creates tables in the database. It is composed of two components: a fixed XSL Style-sheet, named *XSL for ERX Schema*, and a Java class, named *Erx Schema Class*: the former deals with the source XML document, gathers relevant information and passes it to the Java class; this latter one is independent of the actual XML structure, and is focused on the ERX model; furthermore, it provides an abstract interface to the database.

ERX Data Loader. This tool can be viewed as a collection of tools. In effect, the data loading phase depends on the source XML documents to load. Hence, the XSL part is in effect a collection of different style-sheets, one for each document class to load. As before, these style-sheets gathers data to load and call the Java class named *ERX Loader Class*, which provides basic loading primitives. Observe that the set of style-sheets evolves, depending on the evolution of document classes which the system has to deal with. Notice that the adopted solution makes the style-sheet part decoupled w.r.t. the database; furthermore, the Java class is not aware of the actual structure of documents to load.

ERX Query Engine. This tool interprets the ERX Query Language. Again, it is composed of two parts: the XSL part and the Java part. It is clear that the Java part actually performs SQL queries on the underlying database, while the XSL part is responsible for interpreting ERX-QL constructs, thus decoupoling the *ERX Query Class* and the syntactic structure of ERX-QL. However, query interpretation is a more complicated task, w.r.t. ERX Schema processing and data loading;.

4.4 ERX-QL Interpretation

The interpretation of ERX-QL required a particular effort, that led us to define an advanced solution. XSL is very powerful, since it allows a declarative programming

of rather complex document manipulations or transformations. This is a good opportunity to deal with ERX-QL constructs, which are numerous. However, the input for XSL constructs is the input document, in this case the particular ERX-QL Query. However, the *OUTPUT* part of the *FOR-EACH* ERX-QL construct must be repeated for all entity instances selected by the *SELECT* part. Since these instances are obtained by a SQL query performed by the Java class named *ERX Query Class*, a description of these entity instances is not available in the source document (the ERX-QL specification).

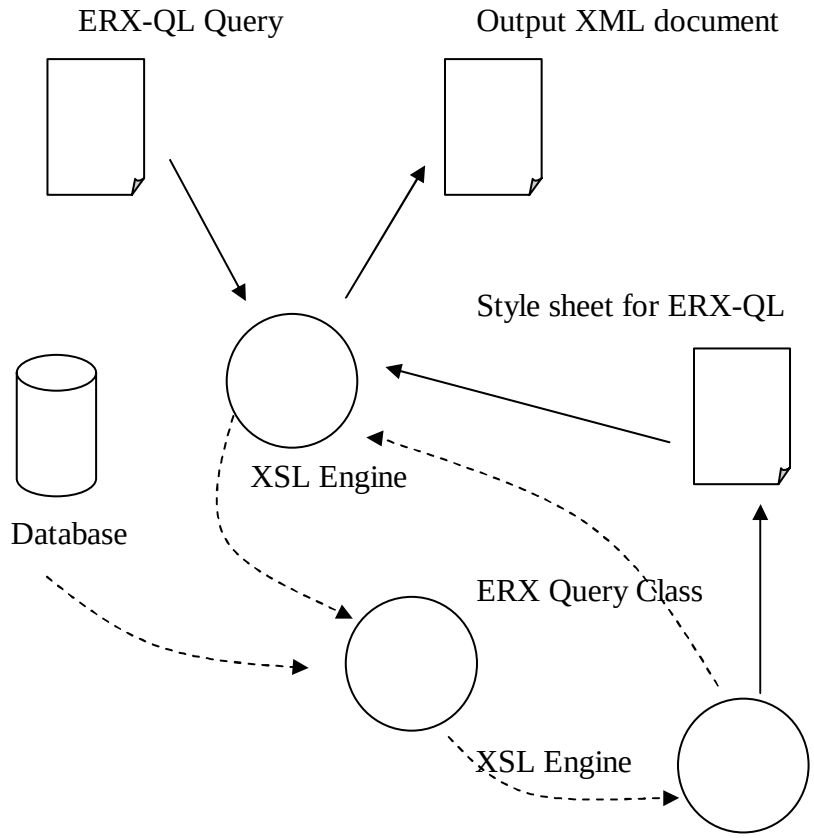


Fig. 4. The ERX-QL interpretation process

This situation was discouraging us, but we wanted to maintain the basic requirement that only XSL style-sheets had to deal with XML syntax. Hence, we adopted the following solution, allowed by the SAXON engine.

The initial instance **I_0** of the SAXON interpreter is activated. It applies the XSL Style-sheet **S** (denoted as *XSL for ERX Query* in the architecture) to the source query **Q**. The XSL Style-sheet **S** finds a *FOR-EACH* construct; it provides to the *ERX Query Class* **C** the description of the *SELECT* part. Then, it extracts the DOM sub-

tree **t_1** corresponding to the *OUTPUT* part of the processed *FOR-EACH* construct. It passes the sub-tree **t_1** to **C**, yielding the control.

The *ERX Query Class* **C** receives the sub-tree **t_1** and actually executes the SQL query on the database. **C** uses both the sub-tree **t_1** and the result set of the SQL query to produce a new DOM document tree **T_1**, obtained joining the content of **t_1** (the *OUTPUT* part to process) and a set of newly generated nodes that describe the result set (these nodes are based on a set of XML elements not present in ERX-QL). Finally, **C** activates a new instance **I_1** of the SAXON interpreter, that interprets the style-sheet **S** on the new document tree **T_1**. The control is yielded to **I_1**.

The current SAXON activation has available a document with both the *OUTPUT* part to process and the result set of the selection part, then the *OUTPUT* part can be correctly interpreted. If the *OUTPUT* part contains another instance of the *FOR-EACH* construct, this mechanism is repeated from step 2, nesting a new SAXON activation (**I_2**), etc..We find the adopted solution very interesting from the technical point of view. In fact, we maintained our choice of processing XML constructs only by means of XSL, so that the ERX Query Class remains unaware of the XML syntax.

Furthermore, we demonstrated that the SAXON interpreter is a very flexible instrument, that allows to exploit the declarative programming style of XSL in complex contexts. Figure 4 describes the ERX-QL Interpretation process.

5 Conclusions

In this paper, we reported about our experience concerning the implementation of the ERX Data Management System. The system provides functionality to gather data from XML documents, store them into a database and then retrieve data to generate new XML documents. The interface of the system is based on the ERX (Entity Relationship for XML) data model and on the ERX Query Language. We motivated the choice for an extended entity relationship data model with the wish to be independent of the particular data model provided by the underlying database. The ER approach also is effective to put in evidence concepts present in data described by XML documents and is independent of any specific XML document class. Hence, by means of ERX it is possible to obtain a view of collected data that is not easily obtainable by maintaining the original tree structure. The ERX Query Language has been designed to navigate the ERX database, select relevant entity instances and generate XML documents. ERX-QL allows recursive queries and the possibility of building query libraries, a means to easily build complex query.

References

1. Bray, T., Paoli, J., Sperberg-McQueen, C. M.: Extensible Markup Language (XML), Technical Report PR-xml-971208, World Wide Web Consortium, December 1997

2. McHugh, J., Widom, J.: Query Optimization for XML, Proc. 25th VLDB Conference, Edinburgh, Scotland, September 1999
3. Tamino XML Database, http://www.softwareag.com/tamino, Software AG
4. Psaila, G. ERX-QL: Querying an Entity-Relationship DB to Obtain XML Documents, Proceedings of DBPL-01 Intl. Workshop on Database Programming Languages, Monteporzio Catone, Rome, Italy, September 2001
5. Kay, M. XSLT Programmer's Reference, Wrox Press, 2000