XML-GL: a graphical language for querying and restructuring XML documents [1]

Stefano Ceri [a,*,2], Sara Comai [a,2], Ernesto Damiani [b,3], Piero Fraternali [a,2], Stefano Paraboschi [a,2], Letizia Tanca [a,2]

[a] *Politecnico di Milano, Dipartimento di Elettronica e Informazione, Piazza Leonardo da Vinci 32, I-20133 Milan, Italy*
[b] *Università di Milano, Polo di Crema, Via Bramante 65, Crema (CR), Italy*

Abstract

The growing acceptance of XML as a standard for semi-structured documents on the Web opens up challenging opportunities for Web query languages. In this paper we introduce XML-GL, a graphical query language for XML documents. The use of a visual formalism for representing both the content of XML documents (and of their DTDs) and the syntax and semantics of queries enables an intuitive expression of queries, even when they are rather complex. XML-GL is inspired by G-log, a general purpose, logic-based language for querying structured and semi-structured data. The paper presents the basic capabilities of XML-GL through a sequence of examples of increasing complexity. © 1999 Published by Elsevier Science B.V. All rights reserved.

Keywords: XML; Query languages; Graphical queries

1. Introduction and motivations

XML [19] is a recent recommendation of the World Wide Web Consortium for a meta-language to define mark-ups for content publishing on the Web. The design goals of XML are driven by the five-year experience of usage of HTML as a content description language, which has exposed several inadequacies:

- the HTML tag set is fixed, and its extension to cover new application requirements either breaks the standard or demands a long standardization process.
- HTML mark-up intermixes structural and visual annotations, producing documents which are hard to process by software agents searching for information on the Web.

XML addresses both problems by letting content producers define and use the set of tags that best mirrors the structure and conceptual properties of the content they want to publish.

The shift from HTML to XML brings a major change in the structure of Web information, which becomes more and more a collection of semi-structured objects, i.e., pieces of content for which at least a partial representation of structure (known as *schema*) is available. This evolution brings forth the necessity of novel languages for extracting informa-

* Corresponding author.
[1] The work presented in the paper has been supported by Esprit Project nr. 28771 'W3I3', and MURST project 'Interdata'.
[2] E-mail: {ceri,comai,fraterna,parabosc,tanca}@elet.polimi.it
[3] E-mail: edamiani@crema.unimi.it

tion from XML content, much in the same way as traditional query languages (notably SQL) have been used for extracting information from structured data stored in databases. As in database applications, the purpose of a query language is twofold:

- letting users extract information from data repositories;
- restructuring information stored in one or more repositories to match novel users' needs.

XML-GL addresses both issues, by permitting the formulation of queries for extracting information from XML documents and for restructuring such information into novel XML documents.

The originality of XML-GL with respect to other proposals for querying XML documents, like XML-QL [6], XQuery [5], XQL [8,16], and XSL [17], is that queries are formulated visually, using a graph-based formalism close to the structure of XML documents (e.g., comparable to the visual representations of XML documents offered by XML authoring tools like the Near & Far XML editor). However, XML-GL is not a visual interface over a conventional, textual, query language, but a graph-based query language with both its syntax and semantics defined in terms of graph structures and operations [2–4,14].

1.1. Requirements for an XML query language

Prior to introducing the features of XML-GL, we propose a set of requirements for the design of a query language over XML documents:

(1) The language should be flexible enough to allow *query formulation both for valid and well-formed documents*. Availability of a DTD should result in a facilitation of both expressing and evaluating a query in XML-GL.
(2) We want to address *queries to a whole Web site*, taking into account links between different documents, rather than querying only one document at a time.
(3) We want the possibility to *access DTDs* and *XML-based metadata* (such as those proposed in the RDF draft standard proposal [1] as well as data, by using the same query paradigm.
(4) We want to *extract* information from an XML document, by means of powerful, yet declarative, pattern matching and element manipulation primitives.
(5) We want to *reshape* the source XML documents, by specifying new links between existing elements as well as new elements in the XML document resulting from a query. This last process requires the introduction of new tags in the result document.
(6) We want to specify *regular expressions on paths*, i.e., the possibility of following recursive and arbitrarily long paths in a site, possibly specifying conditions on path nodes. This feature is particularly helpful in case the search is directed towards certain document patterns, that may be placed arbitrarily in the XML source.
(7) We also want to support arbitrary computations on the numeric content of documents by means of built-in functions.
(8) Given that XML is positional, we may want to allow an *order-sensitive query interpretation*, i.e., one in which the relative order of appearance of XML tags and character data is meaningful. However, such an interpretation may be overly restrictive, so the query language should also offer an unordered interpretation (probably as default).
(9) Under a given query interpretation, XML-GL should be able to compute approximate results, *similar* to the ones that fully satisfy the queries.
(10) Finally, we want queries in our language to be *easy to understand*, even by inexperienced users, and the DTD of the result of the query to be immediately apparent.

2. The XML-GL data model

An XML document can be compliant to a Document Type Definition (DTD), that specifies the types of mark-up elements that can appear in the document, their attributes and containment relationships. If an XML document adheres to a DTD, it is said to be *valid*. If an XML document lacks a DTD but respects some syntactic rules for tag placement, it is said to be *well formed*.

For coherence with the visual nature of XML-GL, we introduce an explicit data model for XML documents, called XML-GDM (XML Graphical Data Model), which we use to represent both the expected

```
<!ELEMENT order (shipto, contact?, item+, date)>
<!ATTLIST order number PCDATA #REQUIRED>
<!ELEMENT shipto (fulladdress|reference)>
<!ELEMENT contact (reference|PCDATA)>
<!ELEMENT fulladdress(company?,city,addressline+)>
<!ELEMENT reference EMPTY>
<!ATTLIST reference customer IDREF>
<!ELEMENT person(firstname?,lastname,fulladdress)>
<!ATTLIST person id ID>
<!ELEMENT company PCDATA>
<!ELEMENT addressline PCDATA>
<!ELEMENT city PCDATA>
<!ELEMENT date (day, month, year)>
<!ELEMENT day PCDATA>
<!ELEMENT month PCDATA>
<!ELEMENT year PCDATA>
<!ELEMENT item(book,quantity,discount?)>
<!ELEMENT book(isbn,title?,price,author*)>
<!ELEMENT author(firstname?,lastname)>
<!ELEMENT firstname PCDATA>
<!ELEMENT lastname PCDATA>
<!ELEMENT isbn PCDATA>
<!ELEMENT title PCDATA>
<!ELEMENT price PCDATA>
<!ELEMENT quantity PCDATA>
<!ELEMENT discount PCDATA>
```

Fig. 1. DTD of the running example.

structure of XML documents (i.e., their DTDs) and actual documents. XML-GDM syntax will be used also (with a few additional graphical notations) for writing XML-GL queries and for representing the DTD of their result, with the benefit of reducing to the minimum the notations that the user should learn to query XML documents.

Well-formed documents without a DTD can be queried in the same way as valid documents: however, since no information about the structure of the document is available at query formulation time, it is more likely that queries may not match exactly the structure of the document and result in empty answers.

2.1. Running example

Consider the DTD shown in Fig. 1, which specifies the structure of documents containing book orders. For each order, the information about the consignee, the ordered items, the date, and possibly a contact address are given. Each item contains the information about the book, the quantity, and possibly the discount percentage. For each book, the ISBN code, the price, and possibly the title and the authors are known. A possible XML document conforming to this DTD is shown in Fig. 2.

2.2. XML graphical data model

The XML-GDM data model consists of three concepts: objects, relationships, and properties:

- *Objects*, depicted as rectangles, indicate abstract items without a directly representable value.
- *Properties*, depicted as circles connected to the object they refer to, indicate representable values (e.g., a character data or parsed character data string); properties have a name and a type, represented as labels.
- *Relationships*, depicted as arcs between objects, indicate semantic associations (e.g., containment or reference). Relationships have an orientation, from a source object to a destination object.

2.3. Representing DTDs and documents

For the sake of explanation, we classify XML content into four categories:

- *Printable content*: PCDATA content [4].
- *Non-terminal elements*: XML elements which include other sub-elements.
- *Terminal elements*: XML elements with printable content or EMPTY content.
- *Mixed elements*: XML elements with either printable content or element content, in mutual exclusion.

XML attributes are also classified as *object-identifiers*, if their type is `ID`, *object-valued*, if their type is `IDREF` or `IDREFS`, or *printable* attributes, otherwise.

The correspondence between an XML DTD and an XML document and an XML-GDM graph is established by the following rules [5]:

[4] For simplicity, in the rest of the paper we will not distinguish between `CDATA` and `PCDATA` content.

[5] The rules assume that all the *parameter entities* used in the DTD (e.g., to represent sub-elements or attributes shared by several elements) have been expanded.

```
<?xml version="1.0" standalone="no"
      encoding="UTF8"?>
<DOCTYPE ORDER SYSTEM "order.dtd">

<ORDER number=1>
 <SHIPTO>
  <REFERENCE customer="C00001"></REFERENCE>
 </SHIPTO>
 <CONTACT>Tim Bell</CONTACT>ITEM>
 <DATE><DAY>14</DAY><MONTH>11</MONTH>
       <YEAR>1998</YEAR></DATE>
 <ITEM>
  <BOOK><ISBN>15536455</ISBN>
    <TITLE>Introduction to XML</TITLE>
    <PRICE>24.95</PRICE>
    <AUTHOR><FIRSTNAME>Charles</FIRSTNAME>
         <LASTNAME>Porter</LASTNAME></AUTHOR>
  </BOOK>
  <QUANTITY>6</QUANTITY>
  <DISCOUNT>.40</DISCOUNT> </ITEM>
 <ITEM>
  <BOOK><ISBN>15532155</ISBN>
    <TITLE>Introduction to Internet</TITLE>
    <PRICE>22.50</PRICE>
    <AUTHOR><FIRSTNAME>Steve</FIRSTNAME>
      <LASTNAME>Andrews</LASTNAME></AUTHOR>
  </BOOK>
  <QUANTITY>10</QUANTITY>
  <DISCOUNT>.42</DISCOUNT> </ITEM> </ORDER>

<ORDER number=2>
 <SHIPTO>
  <FULLADDRESS><COMPANY>ASA</COMPANY>
   <CITY>Los Angeles</CITY>
   <ADDRESSLINE>18 Harvard str.</ADDRESSLINE>
  </FULLADDRESS> </SHIPTO>
 <CONTACT>
  <REFERENCE customer="C00002"></REFERENCE>
 </CONTACT>
 <DATE><DAY>20</DAY><MONTH>11</MONTH>
   <YEAR>1998</YEAR></DATE>
 <ITEM>
  <BOOK><ISBN>15536455</ISBN>
    <TITLE>Introduction to XML</TITLE>
    <PRICE>24.95</PRICE>
    <AUTHOR><FIRSTNAME>Charles</FIRSTNAME>
            <LASTNAME>Porter</LASTNAME>
    </AUTHOR>  </BOOK>
  <QUANTITY>6</QUANTITY>
  <DISCOUNT>.40</DISCOUNT> </ITEM>
 <ITEM>
  <BOOK><ISBN>15532155</ISBN>
    <TITLE>Introduction to Internet</TITLE>
    <PRICE>22.50</PRICE>
    <AUTHOR><FIRSTNAME>Steve</FIRSTNAME>
            <LASTNAME>Andrews</LASTNAME>
    </AUTHOR>  </BOOK>
  <QUANTITY>10</QUANTITY>
  <DISCOUNT>.42</DISCOUNT> </ITEM> </ORDER>

<PERSON id="C00001">
 <FIRSTNAME>Robert</FIRSTNAME>
 <LASTNAME>Moore</LASTNAME>
 <FULLADDRESS><COMPANY>ABC</COMPANY>
   <CITY>Los Angeles</CITY>
   <ADDRESSLINE>10 Michigan str.</ADDRESSLINE>
 </FULLADDRESS> </PERSON>
<PERSON id="C00002">
 <FIRSTNAME>Tom</FIRSTNAME>
 <LASTNAME>Smith</LASTNAME>
 <FULLADDRESS><COMPANY>ASA</COMPANY>
   <CITY>Los Angeles</CITY>
   <ADDRESSLINE>18 Harvard str.</ADDRESSLINE>
 </FULLADDRESS> </PERSON>
<PERSON id="C00003">
 <FIRSTNAME>Steve</FIRSTNAME>
 <LASTNAME>Andrews</LASTNAME>
 <FULLADDRESS><CITY>San Francisco</CITY>
  <ADDRESSLINE>15 Washington str.</ADDRESSLINE>
 </FULLADDRESS> </PERSON>
```

Fig. 2. Running example of XML document.

- Each *non-terminal element* E is mapped to an XML-GDM object with the same name as E.
- Between any two non-terminal elements E_1 and E_2 such that E_1 is a *sub-element* of E_2 we establish a relationship from the object that represents E_2 to the object that represents E_1.
- When a *terminal element* E_1 is a sub-element of another element E_2, it is mapped to a property of E_2 named E_1, with `PCDATA` type. In the representation of an element occurrence in a document, this property has the same value as the XML `PCDATA` content. If a terminal element E is *not* contained in any other element, it is represented as an XML-GDM object, named E, with one (predefined) property named `content`, of type `PCDATA`[6].
- *Element disjunction* (|): if a non-terminal element E contains sub-elements $E_1, \ldots, E_n$ that are in exclusive 'or', i.e., only one of them can be

[6] In the sequel we will omit from representation the type label in case of `PCDATA` and `ID`.

present in E, then an arc is drawn which crosses the relationships between E and $E_1, \ldots, E_n$ labeled 'xor'.

- *Mixed elements*: a mixed element E_1 containing either PCDATA or a sub-element E_2 is represented as an object E_1 including the disjunction of E_2 and the predefined property content, introduced above. This notation considers the predefined property content as equivalent to a predefined element (e.g., named PCDATA) with a single property named content of type PCDATA.
- Each *printable attribute* and *object-identifier* of an element E is mapped to a property of the object that represents E, with the same name and type of the XML attribute. For distinguishing XML attributes from nested terminal elements, for the former we color in black the small circle of the property.
- *Object-valued attributes*: each object-valued attribute of an element declaration E in a DTD is mapped to a relationship from the object that represents E to a predefined XML-GDM object, named ANY, which represents any XML element (terminal or non-terminal). The attribute name in the DTD is mapped to the label of the relationship. In an actual XML document, each object-valued reference from an occurrence of an element E is mapped into a relationship from this occurrence to the single element occurrence having the ID specified in the IDREF attribute of the instance of E. If the object-valued attribute is of type IDREFS, a set of such relationships is introduced.
- *Content model cardinality constraints*: the *iteration* operators '$+$', '$*$' and the *optionality* operator '?' used in the content model of DTDs are expressed as cardinality constraints on either relationships or properties; cardinality constraints have the following forms: $(0:\mathrm{N})$ for $*$, $(0:1)$ for ?, and $(1:\mathrm{N})$ for $+$. If no operator is present, the cardinality constraint $(1:1)$ is assumed. With object-valued attributes, the type IDREF corresponds to the cardinality $(1:1)$ — or $(0:\ 1)$ if the attribute is IMPLIED — while the type IDREFS corresponds to the default cardinality $(1:\mathrm{N})$, — or $(0:\mathrm{N})$ if the attribute is IMPLIED.
- *Element order*: the actual or required order of appearance of sub-elements in a super-element is represented by ordering the arcs that represent the containment relationships counter-clockwise, starting from the arc corresponding to the first sub-element, which is marked by a small trait.

According to the above rules, the DTD of Fig. 1 is represented in XML-GDM as shown in Fig. 3. XML-GDM support the representation of actual XML documents with the same formalism as for DTDs, except that cardinality constraints and disjunctions need to be represented, since a particular element occurrence always has a specific set of sub-elements and particular choice of alternatives in the representation.

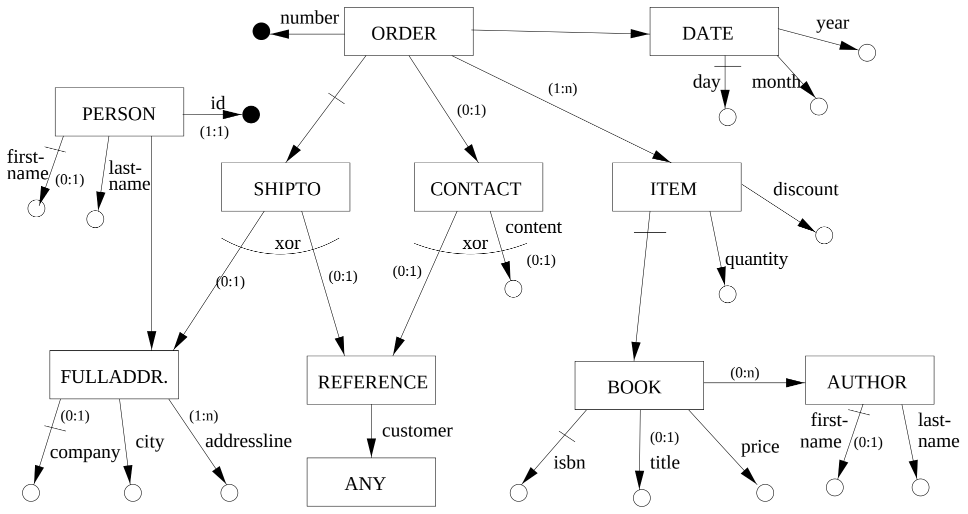

Fig. 3. Example of DTD according to the XML-GML model.

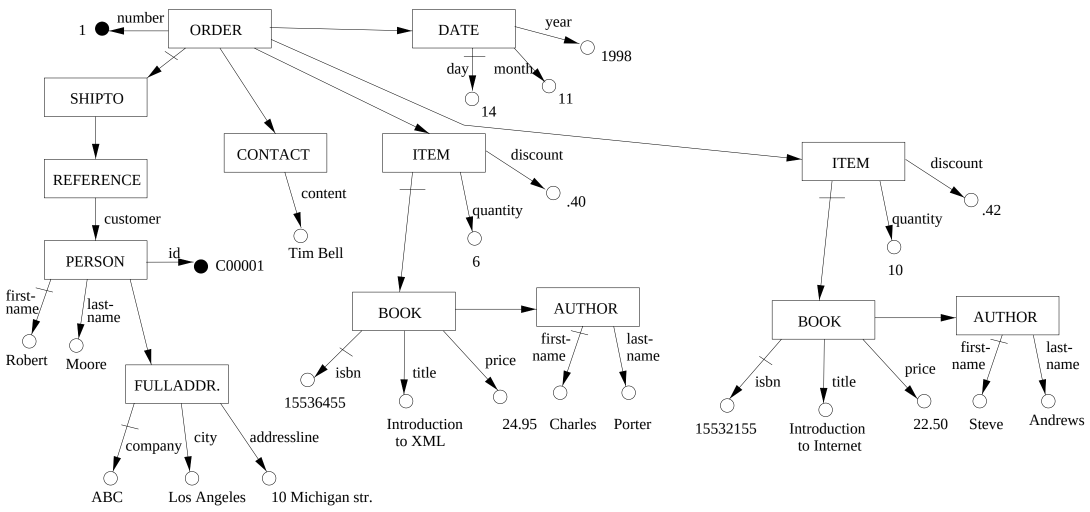

Fig. 4. Representation of the document of the running example in XML-GDM.

A piece of the instance of Fig. 2 is represented in Fig. 4.

3. Query language

XML-GL is a query language for XML-GDM data. An XML-GL query can be applied either to a single XML document or to a set of documents, e.g., those composing a Web site. The query produces a new XML document as the result. Thus, the execution of a query results in a transformation of the source XML document(s) into a new XML document. An XML-GL query consists of four parts:

(1) The *extract* part identifies the scope of the query, by indicating both the target documents and the target elements inside these documents; by drawing a parallel with SQL, the extract part can be seen as the counterpart of the from clause, which establishes the relations targeted by the query.
(2) The *match* part (optional) specifies logical conditions that the target elements must satisfy in order to be part of the query result; continuing the parallel with SQL, the condition part can be seen as the counterpart of the where clause, which chooses the target tuples that are part of the result.
(3) The *clip* part specifies the sub-elements, of the extracted elements that satisfy the match part, to be retained in the result. With respect to SQL, the clip part corresponds to the select clause, which permits the user to define which columns of the result tuples should be retained in the final output of the query.
(4) The *construct* part (optional) specifies the *new* elements to be included in the result document and their relationships to the extracted elements; the same query can be formulated with different construction parts, to obtain results formatted differently. With respect to SQL, the construct part can be seen as the extension of the `create view` statement, which permits the user to design a new relation from the result of a query. The construct part permits both the creation of new elements, the definition of new links, and the restructuring of information local to a given element.

Graphically, an XML-GL query is a pair of XML-GDM graphs, displayed side by side and separated by a vertical line; the left-side graph visually represents the extract and match parts, while the right-side graph conveys the clip and construct parts. This separation sharply evidences those concepts which are used to extract elements from the target documents and those concepts which are used to construct the result documents produced by the query. Indeed, *the right-side graph represents the DTD of the result*.

In the following sections, we progressively introduce the features of XML-GL by means of queries with increasingly complex structures. First, we will show simple queries that extract elements from target documents and produce result documents in different ways (extract-clip queries, Section 3.1); then, we show queries that apply filtering conditions to the extracted elements to be included in the result (extract-match-clip queries, Section 3.2); finally we will introduce queries that define arbitrarily structured result documents, composed of new elements, possibly intermixed with elements extracted from the target documents (extract-match-construct-clip queries, Section 3.3).

3.1. Extract-clip queries

The simplest form of XML-query is the extract-clip, which extracts a portion of an XML document and produces as output a new document containing the extracted data.

Example 1. The query of Fig. 5 *finds all the* `BOOK` *elements from a specified set of documents over the Web*. Its result is shown in Fig. 7b.

In extract-clip queries, the left-side graph contains the extract part of the query. In the example, the extract part operates on the single target element *book*. More generally, the extract part of a query may contain several target elements, represented as root nodes of the left-side graph.

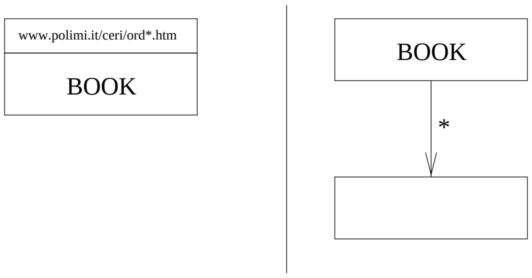

Fig. 5. Example of extract-clip query.

Target elements may optionally contain the indication of the URLs of the document or set of documents that should be used as input in order to evaluate the query. For convenience, URL in queries may contain wildcards; in the query of Fig. 5, the string "http://www.elet.polimi.it/ceri/ord*.xml" inside object `BOOK` makes the query target only those XML documents in the `ceri` directory of host `www.elet.polimi.it`, whose name starts with the string `ord`.

The right-side graph expresses the clip part, which defines the DTD of the result document as an XML-GDM graph, out of the structure of the target elements mentioned in the extract part. When an object mentioned in the extract part should belong to the result of the query, it must be included also in the clip part. The correspondence between left-side and right-side objects is by name (as for object `BOOK` in the example of Fig. 5); if this introduces ambiguity (e.g., for queries using the same elements multiple times in the extract part), then the one-to-one correspondence may be made explicit by drawing an edge that connects the corresponding elements of the two graphs.

The meaning of the one-to-one correspondence is that the result of the query will be constructed-according to the structure dictated by the right-side graph-using exactly those object instances which are selected by the extract part and are mentioned in the clip part. In the example of Fig. 5, all books found in the target XML documents are used to build the result.

When an extracted element is used to build the result, the clip part must also specify which sub-elements should be retained and which should be discarded. To make the clip part more concise, the following shorthand notations are defined, represented in Fig. 6:

- all sub-elements at the first level of nesting are kept (Fig. 6a);

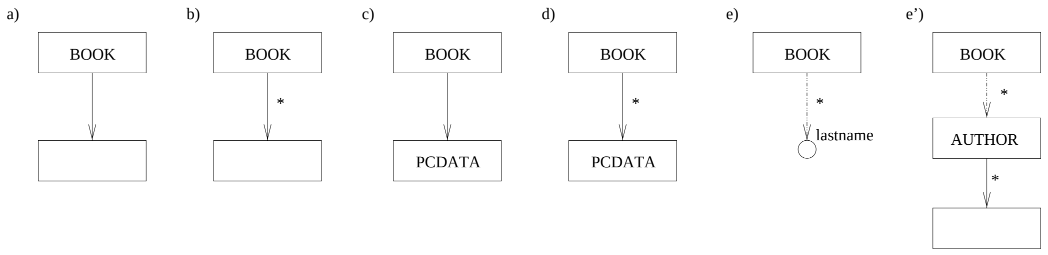

Fig. 6. Graphical notations for expressing the clip.

Notation A: first level elements

```
<BOOK>
  <ISBN>15536455</ISBN>
  <TITLE>Introduction to XML</TITLE>
  <PRICE>24.95</PRICE>
  <AUTHOR></AUTHOR>
</BOOK>
<BOOK>
  <ISBN>15532155</ISBN>
  <TITLE>Introduction to Internet</TITLE>
  <PRICE>22.50</PRICE>
  <AUTHOR></AUTHOR>
</BOOK>
```

Notation B: all level elements

```
<BOOK>
  <ISBN>15536455</ISBN>
  <TITLE>Introduction to XML</TITLE>
  <PRICE>24.95</PRICE>
  <AUTHOR><FIRSTNAME>Charles</FIRSTNAME>
        <LASTNAME>Porter</LASTNAME></AUTHOR>
</BOOK>
<BOOK>
  <ISBN>15532155</ISBN>
  <TITLE>Introduction to Internet</TITLE>
  <PRICE>22.50</PRICE>
  <AUTHOR><FIRSTNAME>Steve</FIRSTNAME>
       <LASTNAME>Andrews</LASTNAME></AUTHOR>
</BOOK>
```

Notation C: first level terminal and `PCDATA`

```
<BOOK>
  <ISBN>15532155</ISBN>
  <TITLE>Introduction to Internet</TITLE>
  <PRICE>22.50</PRICE>
</BOOK>
<BOOK><ISBN>15536455</ISBN>
  <TITLE>Introduction to XML</TITLE>
  <PRICE>24.95</PRICE>
</BOOK>
```

Notation D: all level terminal and `PCDATA`

```
<BOOK>
  <ISBN>15536455</ISBN>
  <TITLE>Introduction to XML</TITLE>
  <PRICE>24.95</PRICE>
  <FIRSTNAME>Charles</FIRSTNAME>
  <LASTNAME>Porter</LASTNAME>
</BOOK>
<BOOK>
  <ISBN>15532155</ISBN>
  <TITLE>Introduction to Internet</TITLE>
  <PRICE>22.50</PRICE>
  <FIRSTNAME>Steve</FIRSTNAME>
  <LASTNAME>Andrews</LASTNAME>
</BOOK>
```

Notation E: all occurrences of a nested property

```
<BOOK>
  <LASTNAME>Porter</LASTNAME>
</BOOK>
<BOOK>
  <LASTNAME>Andrews</LASTNAME>
</BOOK>
```

Notation E': all occurrences of a nested element

```
<BOOK>
  <AUTHOR>
    <FIRSTNAME>Charles</FIRSTNAME>
    <LASTNAME>Porter</LASTNAME></AUTHOR>
</BOOK>
<BOOK>
  <AUTHOR>
    <FIRSTNAME>Steve</FIRSTNAME>
    <LASTNAME>Andrews</LASTNAME></AUTHOR>
</BOOK>
```

Fig. 7. Results of different clip parts for the query of Fig. 5.

- all sub-elements at all levels of nesting are kept (Fig. 6b);
- only the terminal sub-elements and elements of type `PCDATA` at the first level of nesting are kept (Fig. 6c);
- only the terminal sub-elements and elements of type `PCDATA` at all levels of nesting are kept (Fig. 6d);
- all occurrences of a given element found at any level of nesting, without the intermediate enclosing elements, are kept (Fig. 6e and Fig. 6e′).

The results produced by the above clip parts applied to the set of all books in the running example document are illustrated in Fig. 7.

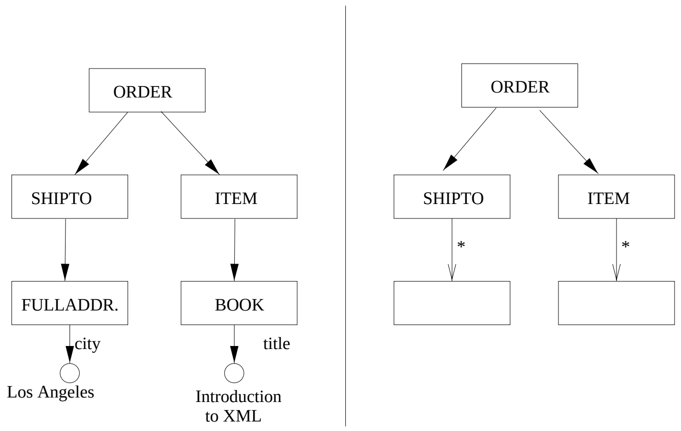

Fig. 8. Example of extract-match-clip query with a match part imposing predicates on sub-elements.

3.2. Extract-match-clip queries

The match part extends the left-side graph of the query with the possibility of expressing a large class of selection predicates. These are described by means of a well-defined collection of graphical notations which enable the expression of existential conditions (e.g., the requirement that a sub-element exists), or predicates on element's properties (e.g., required values for attributes and `PCDATA` content); all predicates are implicitly in conjunctive form.

The condition of a query normally involves several sub-elements of the target elements in the extract part; these elements are evidenced in the left-side graph. This operation is eased by the presence of the XML-GDM representation of the DTD of the input document(s), which exactly gives the needed graphical representation of the elements' internal structure. The same notation is also used to specify the sub-elements to be kept in the clip part, as shown in the previous section. Thus, query construction can be easily supported by a 'drag-and-drop' interface, starting from the construction of the graph in the left part of a query, and then proceeding to the right-side graph.

Example 2. The query of Fig. 8 *finds orders containing the book titled "Introduction to XML", to be shipped to an address in Los Angeles, and presents such orders with their shipping and item information*[7].

The document produced as result of the previous query is the following:

```
<ORDER number=2>
 <SHIPTO>
  <FULLADDRESS><COMPANY>ASA</COMPANY><CITY>Los Angeles</CITY>
   <ADDRESSLINE>18 Harvard str.</ADDRESSLINE>
  </FULLADDRESS>
 </SHIPTO>
 <ITEM>
  <BOOK><ISBN>15536455</ISBN>
   <TITLE>Introduction to XML</TITLE>
   <PRICE>24.95</PRICE>
   <AUTHOR><FIRSTNAME>Charles</FIRSTNAME><LASTNAME>Porter</LASTNAME></AUTHOR>
  </BOOK>
  <QUANTITY>6</QUANTITY>
```

[7] For sake of simplicity, from now on we will omit URLs in the match part.

```
  <DISCOUNT>.40</DISCOUNT>
 </ITEM>
 <ITEM>
  <BOOK><ISBN>15532155</ISBN>
   <TITLE>Introduction to Internet</TITLE>
   <PRICE>22.50</PRICE>
   <AUTHOR><FIRSTNAME>Steve</FIRSTNAME><LASTNAME>Andrews</LASTNAME></AUTHOR>
  </BOOK>
  <QUANTITY>10</QUANTITY>
  <DISCOUNT>.42</DISCOUNT>
 </ITEM>
</ORDER>
```

Note that all orders appearing in the result have both an address and (at least) one item satisfying the extract-match part.

The condition in the match part may also involve the application of boolean operators to attributes and PCDATA properties. To this end, the match part may use the comparison operators (>, <, >=, <=, = and <>) and the string operators (_ and %). The match part can also be used to write queries targeted to several elements, similar to select-join queries of SQL.

Example 3. The query of Fig. 9 *finds all books written bij an author with the same last name as a person whose name starts with 'S'*.

The join condition on last names is expressed in the match part by expanding the graph of the BOOK element to show the inner AUTHOR element, and connecting the author's and person's last names; note that for both the AUTHOR and PERSON elements, lastname is not an XML attribute, but a piece of XML data.

Queries involving element identity and references between elements are easily represented, based on the treatment of element identity explained in Section 2. Element identity is defined when an element has an ID attribute and other elements refer to it using IDREF(S) attributes: in this case, a query may search for IDs that 'point' to the same element, i.e., that have the same value, as demonstrated in the following example.

Example 4. The query of Fig. 10 *finds the persons referenced as contact in an order*. The query extracts those orders containing a contact that includes a reference (order #2, in the running example), and pairs them to the person whose ID matches the *customer* attribute of the reference element. These persons are then included in the result in the clip part.

Element identity can also be used in join conditions: the following query exploits IDREF attributes to 'join' information of orders.

Example 5. The query of Fig. 11 *finds the orders shipped to persons who are also contacts in another*

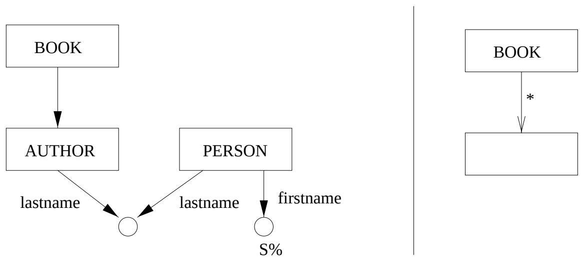

Fig. 9. Example of extract-match-clip query with join.

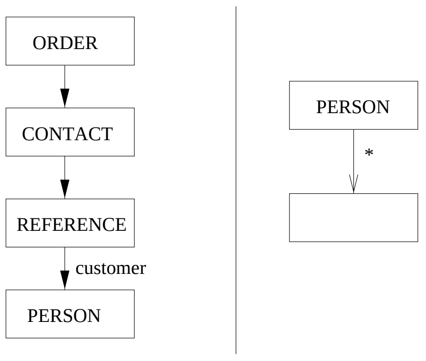

Fig. 10. Example of extract-match-clip query involving ID reference.

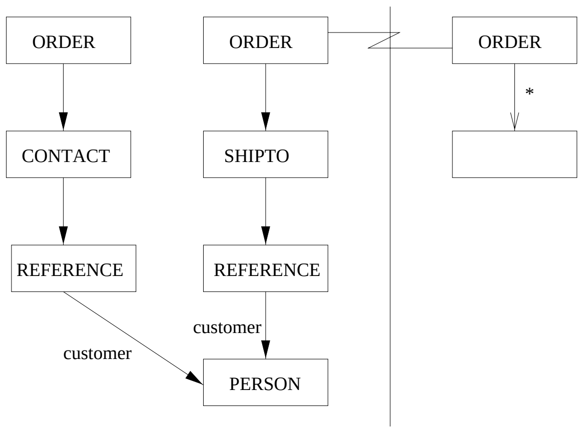

Fig. 11. Example of extract-match-clip query using element identity.

order. Note that in this case the extract-match part of the query contains *two* `ORDER` elements, and ambiguity is avoided by explicitly connecting only one of them to its counterpart in the clip part. This technique compares with the use of alias (with the keyword `as`) in SQL queries, in order to disambiguate multiple occurrences of the same table or column name by associating each of them to a different variable.

Both positive and negated conditions are admitted: to express that a condition is negative, this is drawn using dashed lines, as in the following example.

Example 6. The query of Fig. 12a *finds all books having a title*, while the query of Fig. 12b *finds the books with unknown title*. In the preceding examples, conditions in the match part always specified the name of the element to be extracted/matched: however, if a query requires to express a condition on a generic object, *dummy nodes* are used. They are represented as unlabeled nodes which are matched against `ANY` element.

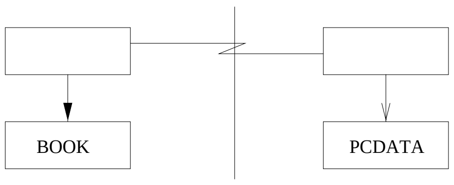

Fig. 13. Example of extract-match-clip query with the special object `ANY`.

Example 7. The query of Fig. 13 *finds all the elements that contain a book and includes them in the result with their* `PCDATA` *content and terminal subelements*.

3.3. Extract-match-construct-clip queries

So far, XML-GL queries have produced very simple result documents, defined by a subset of the elements extracted from target documents in the extract part. However, XML-GL can be used to produce more sophisticated result documents which:

- combine sub-elements of several target elements;
- introduce new elements whose content can be derived from that of existing elements extracted from the target documents;
- contain new elements providing content grouping or reordering capabilities.

From a document processing point of view, the semantics of the construct part of XML-GL queries is similar to that of a *transformation program* that converts a tagged document into another one by means of pattern matching and rewriting, as proposed for instance in the DSSSL (Document Style Semantics and Specification) language [15]. However, while DSSSL provides its transformation language within a stylesheet-based environment for rendering and processing SGML documents, the construct part of XML-GL is concerned with restruc-

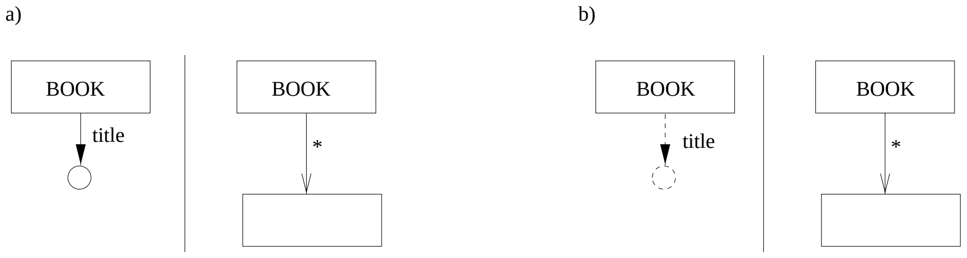

Fig. 12. Examples of extract-match-clip queries with positive (a) and negated (b) conditions in the match part.

turing alone, cleanly separating the transformation from the presentation issues. This corresponds to the functional decomposition envisioned in recent proposals for XML-based Web application environments [12]. Indeed, XML-GL construction can be considered as a complement to current capabilities of XSL (XML Style sheet Language) [17], which again is mainly concerned with XML documents rendering.

3.3.1. Embedding extracted content into new elements

The simplest form of construction consists of embedding elements extracted in the extract-match part into new elements. Three types of embedding are possible:

- *Constructed element*: each element extracted by the extract-match part is embedded into a distinct instance of a new element. Element construction is denoted by a containment relationship between the new element (represented as an XML-GDM object) and its sub-elements in the clip-construct part (see Fig. 14a).
- *List*: all elements extracted by the extract-match part are embedded inside *one* new element. List construction is denoted by a triangle representing the new element connected by a containment relationship to the objects in the clip-construct part representing the sub-elements to be nested (see Fig. 14b).
- *Grouping list*: occurrences of the same element extracted by the extract-match part are embedded inside multiple lists defined by a grouping criterion. Grouping list construction is denoted by an index (a rectangle with horizontal lines) representing the new grouping list connected by a containment relationship to the objects in the clip-construct part representing the sub-elements to be nested (see Fig. 14c). The grouping criterion is represented by an edge connecting the index to one or more elements used for grouping.

Example 8. Consider the three queries of Fig. 14. They find all the existing persons that have an address; with element construction (a) one instance of the new element (called `RESULT`) is created for each person satisfying the given condition and contains the person's data according to the clip specification; with the list construction (b) a single element (also called `RESULT`) is created which contains the list of persons satisfying the match part, along with their nested sub-elements as specified by the clip specification; with the grouping list construction (c) the resulting persons are grouped by city and one element (named `RESULT`) is introduced for each group. Formally, `CITY` induces a partition of the occurrences of `PERSON`, where each distinct value of `CITY` is mapped to a distinct subset of persons.

Thus, the results of the three queries applied to the document of Fig. 2 are the following:

Query A: element construction

```
<RESULT>
 <PERSON id="C00001">
  <FIRSTNAME>Robert</FIRSTNAME>
  <LASTNAME>Moore</LASTNAME>
 </PERSON>
</RESULT>
<RESULT>
 <PERSON id="C00002">
  <FIRSTNAME>Tom</FIRSTNAME>
  <LASTNAME>Smith</LASTNAME>
 </PERSON>
</RESULT>
<RESULT>
 <PERSON id="C00003">
  <FIRSTNAME>Steve</FIRSTNAME>
```

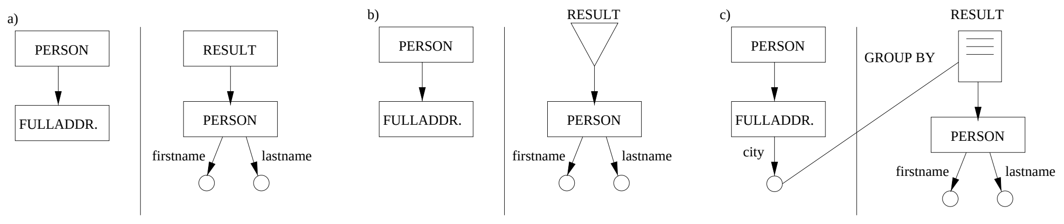

Fig. 14. Examples of constructed element (a), list (b) and grouping list (c).

```
  <LASTNAME>Andrews</LASTNAME>
 </PERSON>
</RESULT>
```

Query B: list construction

```
<RESULT>
 <PERSON id="C00001">
  <FIRSTNAME>Robert</FIRSTNAME>
  <LASTNAME>Moore</LASTNAME>
 </PERSON>
 <PERSON id="C00002">
  <FIRSTNAME>Tom</FIRSTNAME>
  <LASTNAME>Smith</LASTNAME>
 </PERSON>
 <PERSON id="C00003">
  <FIRSTNAME>Steve</FIRSTNAME>
  <LASTNAME>Andrews</LASTNAME>
 </PERSON>
</RESULT>
```

Query C: grouping list construction

```
<RESULT>
 <PERSON id="C00001">
  <FIRSTNAME>Robert</FIRSTNAME>
  <LASTNAME>Moore</LASTNAME>
 </PERSON>
 <PERSON id="C00002">
  <FIRSTNAME>Tom</FIRSTNAME>
  <LASTNAME>Smith</LASTNAME>
 </PERSON>
</RESULT>
<RESULT>
 <PERSON id="C00003">
  <FIRSTNAME>Steve</FIRSTNAME>
  <LASTNAME>Andrews</LASTNAME>
 </PERSON>
</RESULT>
```

Fig. 15 pictorially summarizes the three different construction primitives and how they build the result from the extracted elements. The first option lists all the selected elements, the second option builds a result element containing all the selected persons, and the third option presents them according to the partitioning produced by the grouping criterion.

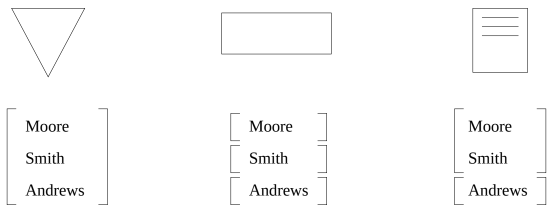

Fig. 15. Summary of contruction primitives.

Example 9. The query of Fig. 16 demonstrates the orthogonal combination of construction primitives; it is a variant of Fig. 14c: it *groups persons by city and embeds these groups into a new element called* `RESULT` *containing also the name of the city*.

The query applied to the document of Fig. 2 returns the following result:

```
<RESULT>
 <CITY>Los Angeles</CITY>
 <PERSON id="C00001">
  <FIRSTNAME>Robert</FIRSTNAME>
  <LASTNAME>Moore</LASTNAME>
 </PERSON>
 <PERSON id="C00002">
  <FIRSTNAME>Tom</FIRSTNAME>
  <LASTNAME>Smith</LASTNAME>
 </PERSON>
</RESULT>
<RESULT>
 <CITY>San Francisco</CITY>
 <PERSON id="C00003">
  <FIRSTNAME>Steve</FIRSTNAME>
  <LASTNAME>Andrews</LASTNAME>
 </PERSON>
</RESULT>
```

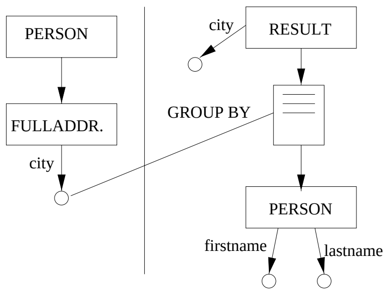

Fig. 16. Example of orthogonal combination of construction primitives.

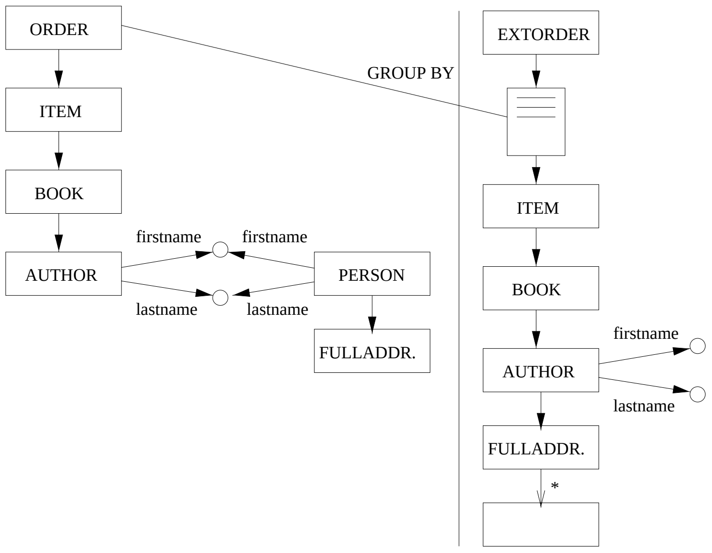

Fig. 17. Example of extension of an element in the contruct part.

3.3.2. Extension of an element

XML-GL can also be used to restructure existing documents, e.g., by including elements of one document into another one or extending the elements of one document with information originating from related elements inside the same document.

Example 10. The query of Fig. 17 *finds the orders containing a book whose author's first name and last name appear also in an element of type* `PERSON` *and produces a new element* `EXTORDER` *where the address is added to each author*. The result of this query applied on the document of Fig. 2 is the following:

```
<EXTORDER>
 <ITEM>
  <BOOK>
   <AUTHOR>
    <FIRSTNAME>Steve</FIRSTNAME>
    <LASTNAME>Andrews</LASTNAME>
    <FULLADDRESS>
     <CITY>San Francisco</CITY>
     <ADDRESSLINE>15 Washington str.
     </ADDRESSLINE>
    </FULLADDRESS>
   </AUTHOR>
  </BOOK>
 </ITEM>
</EXTORDER>
```

In the result, one element `EXTORDER` is constructed for each group of items belonging to the same order retrieved in the extract-match part. Items in the result are the same as those retrieved in the extract-match (as the by-name correspondence indicates), but for a fact: each `AUTHOR` sub-element is extended with the inclusion of the `FULLADDRESS` element coming from the corresponding `PERSON` object retrieved in the extract-match part.

XML-GL offers a rich set of additional features, like unnesting and nesting of XML objects, element ordering, data sorting, arithmetic functions, and aggregate functions. In [2] these additional features are presented, together with a description of the semantics of the language.

4. Related work

The huge amount of data published via the World Wide Web has led to a number of research efforts on techniques to index, query and restructure Web sites contents. In this section we provide a brief overview of related work on XML query languages and, more generally, on query languages for the Web (see also [7]).

A considerable amount of research has been made on how to complement keyword-based *searching* with database-style support for *querying* the Web. Several projects addressed this problem, and three main Web query languages have been proposed so far: Web3QL [9], WebSQL [13] and WebLog [10]. The first two languages are modeled after standard SQL used for relational DBMSs, while the third retains the flavor of the Datalog language.

In the specific domain of XML documents, proposals for query languages are in their infancy. Several preliminary contributions and position papers are collected in [18]. Among the discussed approaches, we review XML-QL [6], XQL [16] by Microsoft, Texcel, and webMethods, XQuery by Inso [5], and XQL [8] by Fujitsu.

The XML-QL language [6] has been submitted for evaluation to the World Wide Web Consortium by a pool of researchers. XML-QL provides a textual syntax for writing queries that construct new XML documents from target documents. The expressive power of XML-QL is comparable to that of XML-GL, but the former has a different syntactic flavor based on the use of pattern-matching expressions and variables ranging over content and tag names to extract content from target documents and embed it into the result of a query.

The XML Query Language (XQL) [16] is a notation proposed by several companies for addressing and filtering the elements and text of XML documents. XQL is an extension to the XSL pattern syntax [7]. The basic idea is to provide a syntax to locate nodes (elements and text) within an XML document, using a notation inspired by directory path expressions. XQL relies on path expressions, filters, and methods to achieve an effect similar to the extract-match part of an XML-GL query. Conversely, there is no counterpart in XQL for the construction of new documents, as provided by the clip-construct part of XML-GL queries.

XQuery [5] is a query language proposed by Inso Corporation for extracting information from XML documents. XQuery draws its syntax from the XPointer document linking language [11]. XQuery provides a rich type system for representing XML content and defines the output of queries as *locations*, which can be sets of XML elements, attributes, or spans of text. A query consists of a sequence of steps: each step selects nodes, either in absolute terms or based on the output of the preceding step. Examples of steps are: “select the element with the given id” or “select from among the direct children of the current location”. Sequences of steps are joined by the dot operator, like in OQL path expressions. XQuery most advanced features address the use of links in queries and of regular expressions to write order-sensitive queries.

XQL [8] by Fujitsu takes a different approach to XML document querying, by proposing a syntax which extends well-known database query languages (SQL and OQL) to address the features of XML data. XQL has a select-from-where construct extended with tag variables, path expressions, and URL specification. XQL also includes primitive for the construction of output documents (inclusive of grouping) comparable to the construct-clip part of XML-GL queries.

5. Conclusions

XML-GL is a sophisticated, but intuitive, visual language for querying XML data sources. It draws its unique features from an original combination of orthogonal, natural primitives for visualizing DTDs and documents, extracting their content, producing new content from extracted data, and formatting query results in complex ways. The use of a visual interface and language for querying XML-based Web documents seems very appealing.

Our research activity will concentrate on the following directions. We will consolidate the language and design a textual version with equivalent expressive power; we have great interest in using a query language jointly designed by the Web community, but a consensual language has to emerge and the language should support our graphical constructs in a natural way. We will also address the requirements not currently satisfied by our proposal, such as queries over arbitrarily linked documents and metadata, and flexible query interpretation and expansion for non-exact document matching. Finally, we will concentrate on the deployment of the language within a Web-based visual environment, by studying an effective query interface supporting the automatic display of DTDs and/or documents and a collection of graphical primitives for clipping and dragging schema elements and for incrementally constructing the query graphs.

References

[1] D. Brickley, R. Guha and A. Layman, W3C RDF Schemas (working draft), October 1998, http://www.w3.org/TR/WD-rdf-schema/.

[2] S. Ceri, S. Comai, E. Damiani, P. Fraternali, S. Paraboschi and L. Tanca, XML-GL: a query language for XML documents, Technical Report 99.6, Dipartimento di Elettronica e Informazione, Politecnico di Milano, February 1999.

[3] S. Comai, E. Damiani, R. Posenato and L. Tanca, A schema-based approach to modeling and querying WWW data, in: Proc. of FQAS'98, Roskilde, Denmark, May 1998, LNAI 1495.

[4] A. Cortesi, A. Dovier, E. Quintarelli and L. Tanca, Operational and abstract semantics of a query language for semi-structured information, in: Proc. Int. Workshop on Deductive Logic Programming (DDLP '98), 1998.

[5] S.J. DeRose, XQuery: a unified syntax for linking and querying general XML documents, in: Query Languages 98.

[6] A. Deutsch, M. Fernandez, D. Florescu, A. Levy and D. Suciu, XML-QL: a query language for XML, in: Proc. QL'98 — The Query Languages Workshop (World-Wide Web Consortium, Query Languages 98, Cambridge, MA, December 1998), http://www.w3.org/TR/1998/NOTE-xml-ql-19980819.

[7] D. Florescu, A. Levy and A. Mendelzon, Database techiques for the World-Wide Web: a survey, ACM Sigmod Record 27 (3) (1998).

[8] H. Ishikawa, K. Kubota and Y. Kanemasa, XQL: a query language for XML data, in: Query Languages 98 (World-Wide Web Consortium, Query Languages 98, Cambridge, MA, December 1998).

[9] D. Konopnicki and O. Shmueli, W3QL: a query system for the World Wide Web, in: Proc. 21th Int. Conf. on Very Large Databases, Zurich, 1995.

[10] L. Lakshmanan, F. Sadri and I. Subramanian, A declarative language for querying and restructuring the Web, in: Proc. RIDE-NDS, IEEE Computer Soc. Press, 1996.

[11] E. Maler and S. DeRose, XML Pointer Language (XPointer), March 1998, http://www.w3.org/TR/WD-xptr.

[12] H. Maruyama, N. Uramoto and K. Tamura, XML, purpose and use in Web applications, 1998, http://www.software.ibm.com/xml.

[13] A. Mendelzon, G. Mihaila and T. Milo, Querying the World Wide Web, in: Proc. Conf. on Parallel and Distributed Information Systems, Toronto, Canada, 1996.

[14] J. Paredaens, P. Peelman and L. Tanca, G-log a declarative graph-based language, IEEE Trans. on Knowledge and Data Eng. 7 (3) (1995) 436–453.

[15] P. Prescod, An introduction to DSSSL, http://itrc.uwaterloo.ca/papresco/dsssl/tutorial.html.

[16] J. Robie, J. Lapp and D. Schach, XML query language (XQL), in: Query Languages 98 (World Wide Web Consortium. Query Languages 98. Cambridge, MA, Dec. 1998).

[17] World-Wide Web Consortium, An introduction to XSL, 1998, http://www.w3C.org/Style/XSL.

[18] World-Wide Web Consortium, Query Languages 98, Cambridge, MA, December 1998.

[19] World-Wide Web Consortium, XML 1.0, February 1998, http://www.w3.org/XML.

Stefano Ceri is professor at the Dipartimento di Elettronica e Informazione, Politecnico di Milano; he has been visiting professor at the Computer Science Department of Stanford University between 1983 and 1990. His research interests are focused on extending database technology to incorporate data distribution, deductive rules, active rules, and object-orientation; he is also currently interested in the integration between Web and database technologies. He is author of several books, including *The Art and Craft of Computing* (Addison-Wesley, 1997), *Advanced Database Systems* (Morgan Kaufmann, 1997), and *Active Database Systems* (Morgan Kaufmann, 1995).

Sara Comai received her Laurea degree in Ingegneria Gestionale in 1996 from Politecnico di Milano (Italy). Since 1997 she is a Ph.D. student in Ingegneria Informatica e Automatica at the same university. Her research interests are mainly in the areas of active databases and semistructured information representation and processing.

Ernesto Damiani holds a Laurea degree in Ingegneria Elettronica from Università di Pavia and a Ph.D. degree in Computer Science from Università di Milano. He is currently an assistant professor at the campus located in Crema of Università di Milano, and a Visiting Lecturer at the Computer Science Department of LaTrobe University in Melbourne, Australia. His research interests include distributed and object oriented systems, semi-structured information processing and soft computing.

Piero Fraternali is an associate professor at the Dipartimento di Elettronica e Informazione of Politecnico di Milano. He received the Laurea Degree in Ingegneria Elettronica in 1989, and a Ph.D. in Ingegneria Informatica in 1994, both from Politecnico di Milano. His main research interest is currently in the area of the integration of Web and databases. His research focuses also on active databases, object orientation, and software engineering methodologies. He is the author, with Stefano Ceri, of the book *Designing Database Applications with Objects and Rules: The IDEA Methodology* (Addison-Wesley, 1997).

Stefano Paraboschi is an associate professor at the Dipartimento di Elettronica e Informazione of Politecnico di Milano. He received the Laurea Degree in Ingegneria Elettronica in 1990, and a Ph.D. in Ingegneria Informatica in 1994, both from Politecnico di Milano. His main research interests are in the area of databases, with a focus on active databases, data warehouses, and the construction of data-intensive Web sites. He is the author, together with Paolo Atzeni, Stefano Ceri, and Riccardo Torlone, of the book *Database Systems: Concepts, Languages and Architectures* (McGraw-Hill, 1999).

Letizia Tanca is professor at the Dipartimento di Elettronica e Informazione, Politecnico di Milano; she has been professor at Università di Verona between 1995 and 1998. Her research interests concern advanced database languages and systems, and currently focus on query languages for the Web, graphical query languages, and active database systems. She is author, with Stefano Ceri and Georg Gottlob, of the book *Logic Programming and Databases* (Springer-Verlag, 1990).