World Wide Web: Internet and Web Information Systems, 8, 287–316, 2005
© 2005 Springer Science + Business Media, Inc. Manufactured in The Netherlands.
DOI: 10.1007/s11280-005-0646-x

XQBE: A Graphical Environment to Query XML Data

DANIELE BRAGA braga@elet.polimi.it
ALESSANDRO CAMPI campi@elet.polimi.it
Politecnico di Milano - Dipartimento di Elettronica e Informazione, Piazza Leonardo da Vinci 32, 20133, Milano, Italy

Published online: 1 August 2005

Abstract

XQuery, the standard query language for XML, is increasingly popular among computer scientists with SQL background, since XQuery and SQL require comparable skills. However, these experts are limited in number, and the availability of easier XQuery "dialects" could be extremely valuable. With this motivation in mind, we designed XQBE, a visual dialect of XQuery inspired by the QBE language (Query by Example). Coherent with the hierarchical nature of XML, XQBE uses one or more hierarchical structures to denote the input documents and one structure to denote the document produced in output. These structures are annotated to express selection predicates; explicit binding edges connecting the nodes of these structures visualize the input/output mappings. This paper presents XQBE through several examples and describes the main features of our implementation of the language, a visual editor coupled with an XQBE-to-XQuery translator. Indeed, the XQBE front-end is a general purpose user-friendly visual query interface, capable of providing access to any data storage system that exposes XQuery APIs. Available schema information can be exploited to guide users in querying data sets they are not familiar with. Also, switching between the visual and textual versions of the same query could be helpful for XQuery learners.

Keywords: XML, query languages, visual interfaces

1. Introduction

The diffusion of XML in many applicative fields sets a pressing need for providing the capability to query XML data to a wide spectrum of users, including those with minimal or no computer programming skills at all. This paper describes a user friendly interface, based on an intuitive visual query language, that we developed for this purpose.

The success of the QBE paradigm [29] demonstrated that a visual interface to a query language is effective in supporting the intuitive formulation of queries when the basic graphical constructs of the language are close to the visual abstraction of the underlying data model. Accordingly, while QBE is a relational query language, based on the representation of tables, XQBE (*XQ*uery *B*y *E*xample) is based on the use of annotated trees, so as to adhere to the hierarchical nature of the XML data model. The syntax and semantics of XQBE are formally defined in [5].

1.1. Motivation and design principles

XQuery [28], promoted by the W3C (World Wide Web Consortium), is the official textual language for querying XML data. This language, however, is far too complicated for occasional or unskilled users, only aware of the basics of the XML data model or simply conscious of the schema of the documents they have to query. Nevertheless, this basic knowledge should be enough to allow any user to formulate queries with a suitably simple (maybe ad hoc) query language. In this paper we assume the reader is familiar with XQuery; if that was not the case, a good example driven introduction is the W3C Use Case [27]. In any case, we trust that our visual queries are readable and comprehensible also to those without a specific XML background.

XQBE was designed with the twofold objective of being intuitive, according to the aforementioned principles, and easy to map directly to XQuery, so as to be a GUI capable of running on top of any existing XQuery engine. XQBE includes most of the expressive power of XPath, allows for arbitrarily deep nesting of XQuery FLWOR expressions (the basic For-Let-Where-Orderby-Return building blocks of the language), supports the construction of new XML elements and permits to restructure existing documents. However, the expressive power of XQBE is limited in comparison with that of XQuery. As an example, XQBE does not support user defined functions, as we believe that a user confident with this abstraction can directly use XQuery; another limitation of XQBE concerns the support for disjunction. These limitations are precise design issues, as we believe that a complete but too complex visual language would fail both in replacing the textual one and in addressing most users' needs.

The particular purpose of XQBE makes *usability* one of its critical success factors, and we therefore kept an eye on this aspect during the whole design and implementation process. In this perspective, the set of visual constructs evolved to the current XQBE syntax, with which we believe we reached a good trade off between the need for a neat graphical characterization (with different constructs for different concepts) and the fact that an unreasonably large set of symbols would be rather confusing.

1.2. Paper organization

Section 2 presents XQBE by means of several examples taken from or inspired by the W3C "XML Query Use Cases" [27], then Section 3 presents our implementation of XQBE, discussing the architecture, how XQBE is translated into XQuery, and how the availability of schema information can benefit the query composition process. The paper concludes with Section 4, where previous related research work is discussed.

2. XQuery by example

In order to introduce the look and feel and the basic features of XQBE we first show the XQBE version of some simple queries, taken from one of the W3C XML Query

```
<bib>

  <book year="1994">
    <title> TCP/IP Illustrated </title>
    <author> <last> Stevens </last>
             <first> W. </first> </author>
    <publisher> Addison-Wesley </publisher>
    <price> 65.95 </price>
  </book>

  <book year="1992">
    <title> Advanced Programming in the Unix...</title>
    <author> <last> Stevens </last>
             <first> W. </first> </author>
    <publisher> Addison-Wesley </publisher>
    <price> 65.95 </price>
  </book>

  <book year="2000">
    <title> Data on the Web </title>
    <author> <last> Abiteboul</last>
             <first> Serge </first> </author>
    <author> <last> Buneman </last>
             <first> Peter </first> </author>
    <author> <last> Suciu </last>
             <first> Dan </first> </author>
    <publisher> Morgan Kaufmann Publishers </publisher>
    <price> 39.95 </price>
  </book>

  <book year="1999">
    <title> The Economics of Technology and... </title>
    <editor> <last> Gerbarg </last>
             <first> Darcy </first>
             <affiliation> CITI </affiliation> </editor>
    <publisher> Kluwer Academic Publishers </publisher>
    <price> 129.95 </price>
  </book>

</bib>
```

Figure 1. A sample document (www.bn.com/bib.xml).

Use Cases [27], the one named “XMP”, which is based on the XML fragment of Figure 1.

2.1. Simple queries

The first query, q1 (named Q1 in [27]), reads “*List books published by Addison-Wesley after 1991, including their year and title*”. In XQuery:

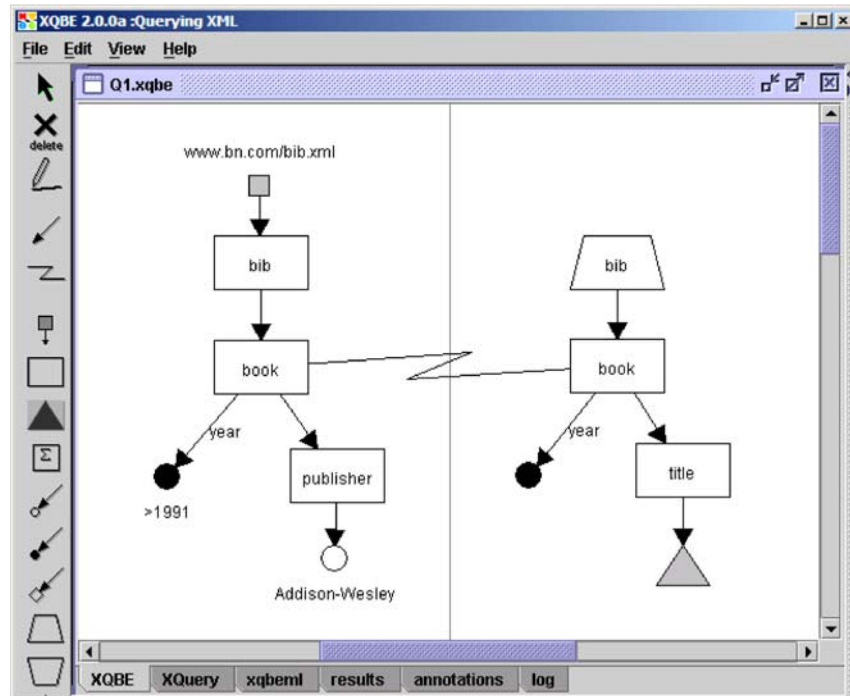

Figure 2. A query with selection conditions (q1).

```
<bib>
 {  for $b in doc("www.bn.com/bib.xml")/bib/book
    where $b/publisher="Addison-Wesley" and $b/@year>1991
    return <book year={$b/@year}>
               { $b/title}
           </book>
 }
</bib>
```

The XQBE version of q1 is shown in Figure 2. A query always has a vertical line in the middle, that separates the *source* part (on the left) from the *construct* part (on the right); so the query has a "natural" reading order from left to right. Both parts contain labeled graphs that represent XML fragments and express properties of such fragments (like conditions upon values, ordering properties, etc.): the source part describes the XML data to be matched in order to construct the query result, while the construct part specifies which parts are to be retained in the result and (optionally) which newly generated XML items are to be inserted. The correspondence between the components of the two parts is expressed by explicit binding edges across the vertical line; these edges con-

nect the nodes of the source part to the nodes that will take their place in the output document.

All the XML *elements* in the source part of the target documents are represented as labeled rectangles; *attributes* are represented as black circles, with the attribute *name* on the arc between the rectangle and the circle; PCDATA content is represented as an empty circle. Black and empty circles together are named *value* nodes. Value nodes may be labeled, so as to express conditions on the values they represent.

The query in Figure 2. extracts data from the document of the running example: the source part matches all the `book` elements with a `year` attribute whose value is greater than 1991 and a `publisher` subelement whose PCDATA content equals “Addison-Wesley”. As shown by the grey node above the `bib` node in the source part, documents are referenced in XQBE by means of square grey root nodes, labeled with a URL to locate the XML documents that are the target of the query.

In the construct part, the paths that branch out of a bound node indicate which of its sub-items are to be retained, thus “projecting” the bound node. In q1 only the title and publication year of the selected `books` are retained. The grey triangular node below the `title` node expresses the inclusion into the result of the entire `title` fragments, obtained by means of projection from each book element. This notation is always used to synthetically include entire fragments in the result.

The binding edge between the `book` nodes states that the query result shall contain *as many* `book` elements *as* those matched in the source part. Rectangular nodes in the construct part represent XML elements that are included in the result in direct correspondence with the extraction of some XML data from the source documents (either by including in the result the extracted elements themselves or by projecting and renaming such elements).

The trapezoidal `bib` node above the book node means that all the generated books are to be contained into one `bib` element. This node represents a newly generated element, and *new elements* are always represented as trapezia in XQBE. Trapezia can have their short edge on the upper side or on the bottom side. These two node types impose different cardinality constraints on the newly generated elements:

- If the short edge is above, one new element is generated to contain all the items corresponding to the nodes reached by the outgoing arcs—i.e. all the subelements will be wrapped by the same tag.
- If the short edge is below, each item corresponding to the sub-nodes is contained into a different instance of the newly generated element—i.e. each instance is wrapped by a different tag.

2.2. Element projection and renaming

Query Q3 in [27] reads “*For each book in the bibliography, list the title and authors, grouped inside a result element*”. In XQuery:

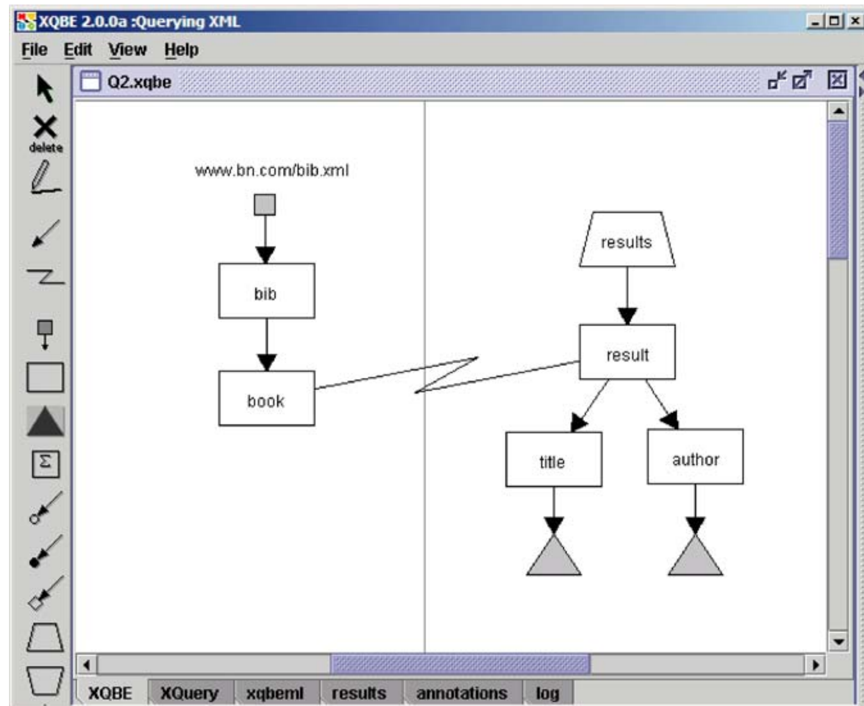

Figure 3. Projection and renaming (q2).

```
<results>
{ for $b in doc("www.bn.com/bib.xml")/bib/book
  return <result>
            { $b/title }
            { $b/author }
          </result>
}
</results>
```

In this query there are no selection conditions, but only a "projection" with "renaming" of all the book elements. In the XQBE version (q2 in Figure 3) the binding edge between the book element and the result element causes the construction of a result element for each book in the source document. The author and title elements below result extract the corresponding subelements of book, thus projecting the book element bound to result.

The previous queries project the book elements "in breadth", but dealing with trees also requires the ability to project them "in depth", i.e. to take far descendants of a given element and place them as direct subelements of that element, pruning the elements in the middle.

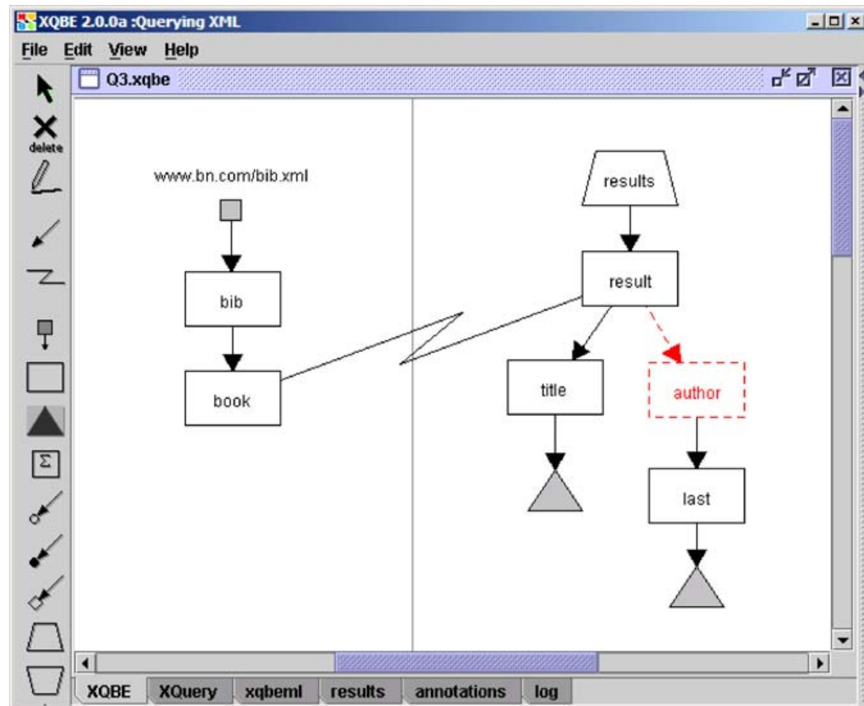

Figure 4. Breadth/depth projection (q3).

As an example consider a query that reads “*For each book list only the title and the surnames of the authors*”. In XQuery:

```
for $b in doc("www.bn.com/bib.xml")/bib/book
return <book>
           { $b/title }
           { $b/author/last }
       </book>
```

In this case the `author` elements are pruned from the generated result, and are represented in XQBE drawing the `author` node in the construct part with dashed lines (see q3 in Figure 4); `last` elements are directly inserted into the `book` elements. Dashed nodes in the construct part are named “ghost” nodes.

2.3. *Join between two documents*

Query q4 (Q5 in [27]) constructs a joint book catalogue, collecting information from different documents. It reads “*For each book found at both bn.com and amazon.com, list the title of the book and its price from each source*”. In XQuery:

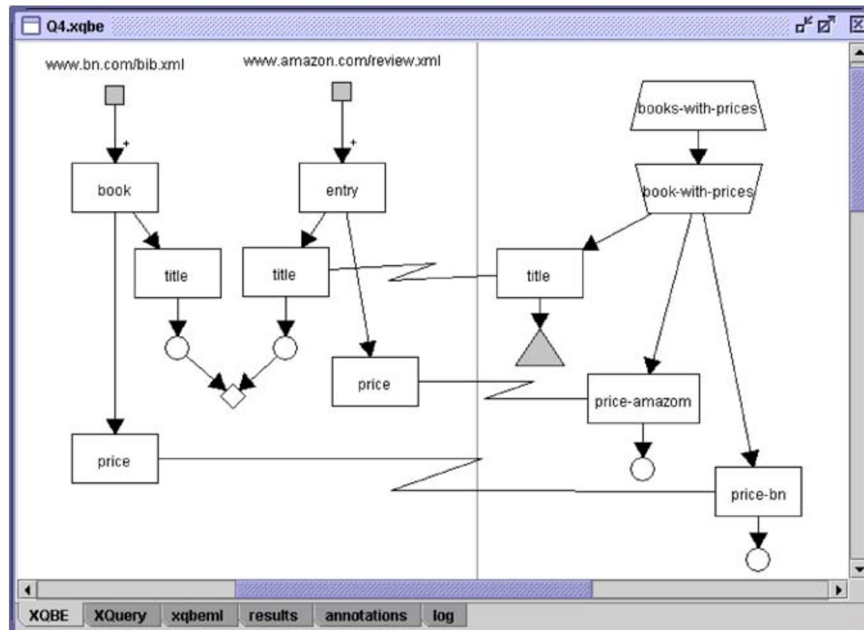

Figure 5. Join between two documents (q4).

```
<books-with-prices>
{ for $b in doc("www.bn.com/bib.xml")//book,
      $a in doc("www.amazon.com/review.xml")//entry
  where $b/title = $a/title
  return
    <book-with-prices>
      { $b/title }
      <price-amazon> { $a/price/text() } </price-amazon>
      <price-bn> { $b/price/text() } </price-bn>
    </book-with-prices>
}
</books-with-prices>
```

This query performs the "inner join" of the books of two documents based on their title. In the XQBE query of Figure 5 the equality between the values is expressed by means of the confluence into a *join* node (a rhombus) of the arcs outgoing from the PCDATA nodes of *both* `title` elements. More generally, the language allows to specify conditions by labeling join nodes with binary predicates (<, <=, !=, ...); equality is the default if unspecified. The `price-amazon` and `price-bn` elements are explicitly bound to include them in the result.

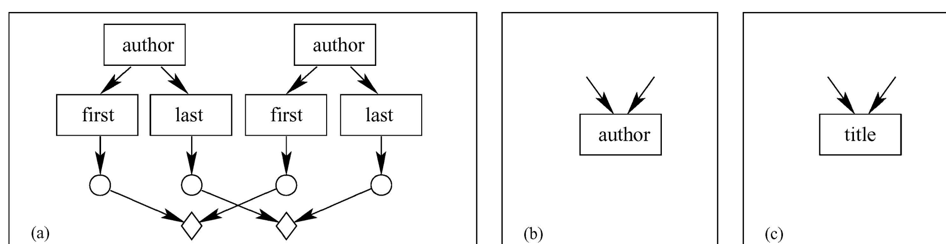

Figure 6. Shortcut for joins.

Also note that `book` and `entry`[1] are not root elements in the documents referenced in the query. The crosses on the arcs outgoing from the root nodes extend the inclusion to all levels of depth; cross on arcs express the transitive closure of the containment relationship, reminding of the Kleene cross, and corresponds to the use of the '//' XPath step in XQuery.

The join in q4 is based on a single value, but XQBE provides a useful and intuitive shorthand for all the cases in which equality is expressed for complex fragments. If we consider a join based on the authors' full name, an exhaustive representation would be that of Figure 6(a), while the compact notation supported by XQBE is that of Figure 6(b) (where one node does the job of the twelve nodes of Figure 6(a)). Figure 6(c) shows that the join of q4 could have exploited the same shorthand, even if the `title` fragment only consists of an element with PCDATA content. The general principle states that whenever a confluence occurs on a rectangular node this requires that the entire fragments that originate from that node are equal. The semantics of this *deep equality* is the same as that of the *deep-equal()* function in XQuery, when applied to elements with complex content.

2.4. *Document restructuring*

XQBE allows to express many kinds of document transformations with the constructs illustrated so far. We now show an example of flattening of hierarchical data structures. Query q5 (Q2 in [27]) reads "*Create a flat list of all the title-author pairs, with each pair enclosed in a* result *element*". In XQuery:

```
<results>
{ for $b in doc("www.bn.com/bib.xml")/bib/book,
      $t in $b/title, $a in $b/author
  return <result>
           { $t }
           { $a }
         </result>
}
</results>
```

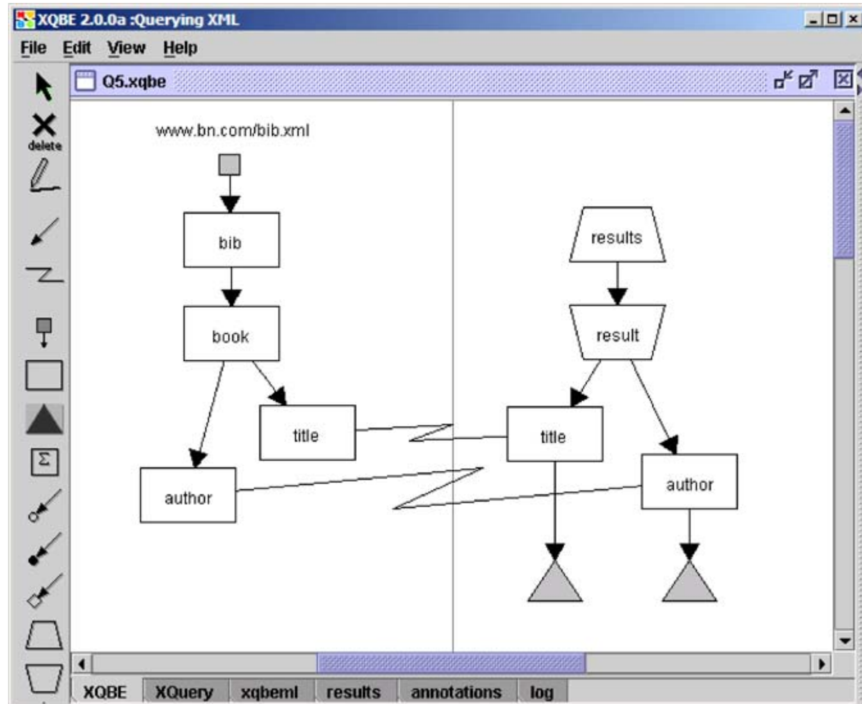

Figure 7. Document restructuring (q5).

Its XQBE version is shown in Figure 7. The `result` element in the construct part has no direct relationship with the book element in the source part. It is a trapezoidal node with the short node below, so its cardinality is determined by the nodes reached by its outgoing arcs. These nodes are bound to nodes with a common ancestor; a `result` element is to be generated for each title-author couple, but not all the couples are to be considered: the "shared" `book` in the source part imposes that only the `titles` and `authors` that are contained into the same `book` are considered. Also note that the correspondence between the `titles` and the `authors` in the match and construct part has to be explicitly declared by means of binding edges, because it cannot be automatically determined.

2.5. *Filtering nodes in the construct part*

Consider the following XQuery statement, that translates the query "*Make a list of all the books with their title, including the editors only if they are affiliated to CITI*":

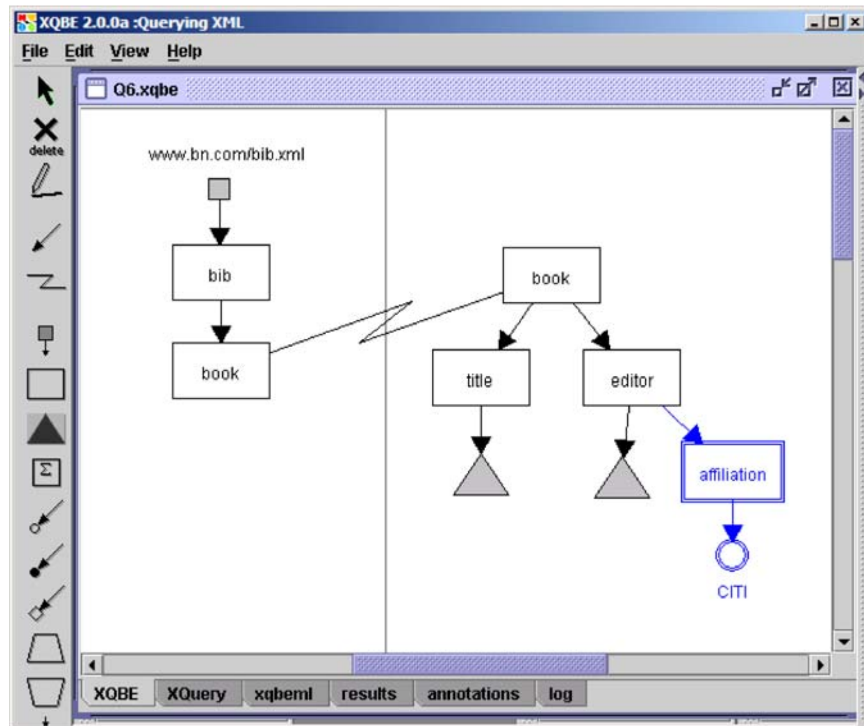

Figure 8. Conditions in the construct part (q6).

```
for doc("www.bn.com/bib.xml")/bib/book
return <book>
          { $b/title }
          { $b/editor[affiliation="CITI"] }
       </book>
```

This query enforces a constraint (affiliation to CITI) that does not intervene in the selection of the matching source data, but only prunes the extracted XML items during the construction of the result. This is typically done in XQuery by placing filters into the path expressions of the `return` clause. In the graphical version the affiliation constraint cannot be specified in the *match* part (this would prune all the books without an editor from CITI), but has to be put in the *construct* part. For this purpose we introduce the notion of *filtering* nodes, represented in double lines (see q6 in Figure 8.). In general, a double lined subtree expresses a restriction that applies to the element in which the subtree is rooted.

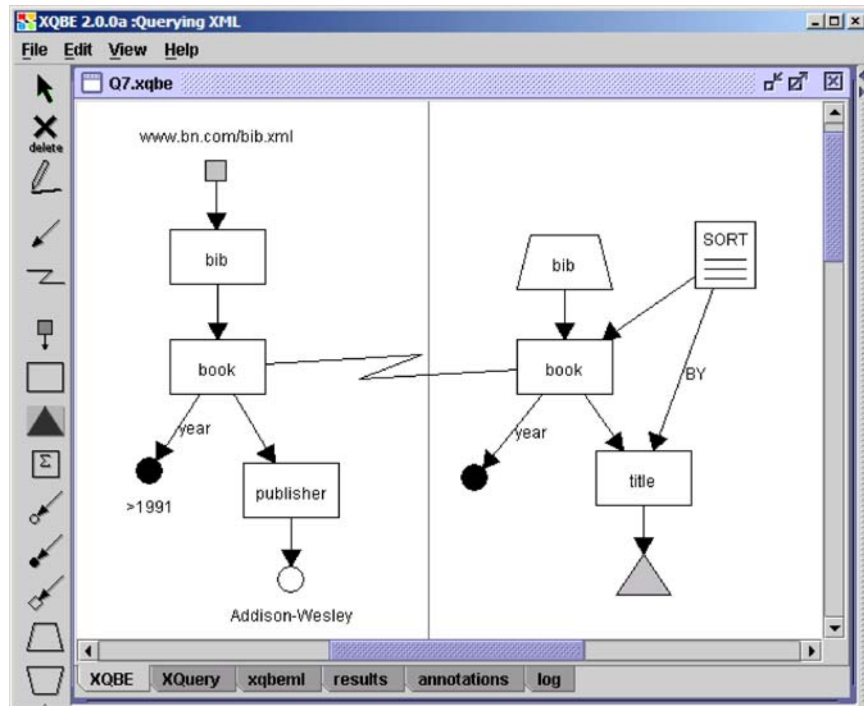

Figure 9. Sorting (q7).

2.6. *Sorting*

Consider query q7 (actually an extension of q1), that reads "*List books published by Addison-Wesley after 1991, including their year and title, sorting the retrieved books in lexicographic order*" (Q7 in [27]):

```
<bib>
 {   for $b in doc("www.bn.com/bib.xml")/bib/book
     where $b/publisher="Addison-Wesley" and $b/@year>1991
     order by $b/title
     return <book>
                { $b/@year }
                { $b/title }
              </book>
}
</bib>
```

The only difference with the XQuery version of q1 is the addition of the `order` by clause. Accordingly, in the graphical representation we only need to add a `SORT` node (see Figure 9).

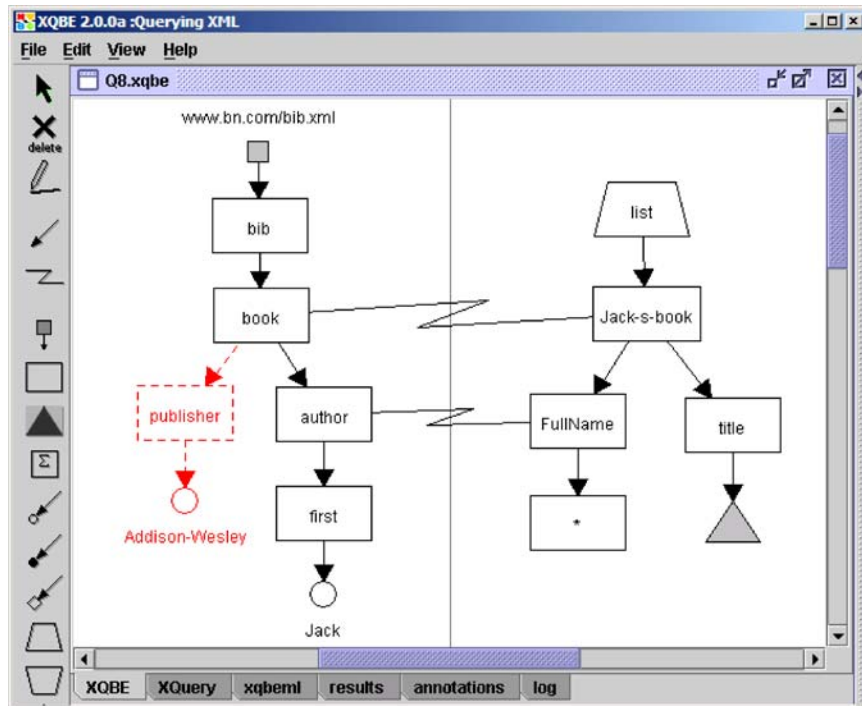

Figure 10. Nesting and negation (q8).

2.7. *Nesting and negation*

As a last example consider the following XQuery statement (the XQBE version is in Figure 10):

```
<list>
 {  for $b in doc("www.bn.com/bib.xml")/bib/book
    where some $a in $b/author satisfies
           some $f in $a/first/text() satisfies
           ($f = "Jack" and
           not(some $p in $b/publisher/text() satisfies
               ($p = "Addison-Wesley")))
    return <Jack-s-book>
           {for $a in $b/author
           where some $f in $a/first/text() satisfies
               ($f = "Jack")
           return <FullName> { $a/* } <FullName> }
```

```
            {$b/title}
            </Jack-s-book>
}
</list>
```

It translates q8, that reads "*List all books not published by Addison-Wesley and with an author whose first name is Jack. Rename each of these books in <Jack-s-book>, and only retain the title and the full name of the authors whose first name is Jack*". The query in Figure 10 contains *negated nodes*, represented in the source part by means of dashed lines, which impose a negative condition to the node they are attached to, and namely the *non*-existence of an XML fragment that matches the negated configuration. In this example we ask for `book` elements inside which no publisher elements exist with a PCDATA content equal to "Addison-Wesley".

Note that the notation for dashed nodes is overloaded: they represent negated conditions when used in the source part and represent elements that are not to be retained in the result when used in the construct part. The meaning is clearly different, but in both cases they match XML items that *must not be contained* in the data. This overlap of notation is intuitive in the opinion of the majority of those people who evaluated our interface.

2.8. XQBE vs. QBE

Due to the major complexity of the XML data model w.r.t. flat relational tables and to the major complexity of XQuery w.r.t. SQL, XQBE is more limited in comparison with XQuery than QBE is in comparison with SQL; however, XQuery is Turing-complete. The major legacy of QBE is its underlying philosophy: the idea of showing an example of a fragment of interest to be matched against the source data set and an example of the structure of the result to be constructed with the matching fragments.

When relational data is canonically converted to XML:

– Selection predicates are put directly on the leaves of the XQBE structures, which represent values.
– Joins in XQBE have the same interpretation and effect than in QBE/SQL (as shown by q4).

All the constraints expressed by the predicates in the source part must hold in conjunction, as we decided that combining the structural complexity with the complexity of predicates would exceed the expectations of the average XQBE users. Also, XQBE has nothing comparable with the QBE "condition box", which is meant to express complex predicates - at least too complex to fit in the visual representation of a table; instead, we decided as a design issue to disallow any constraint that cannot be expressed directly on the query graphs.

Aggregate functions are available as well, but with limited grouping capabilities. In particular, XQBE allows to synthetically and explicitly apply aggregate functions to groups

only if the native nesting structure within the data source reflects the desired grouping criterium.

3. Our implementation of XQBE

XQBE is fully implemented in a tool environment based on a client-server architecture; the implementation consists of about 120 Java classes. Along with the core capabilities of drawing XQBE queries and generating their translation into XQuery, we have also included a considerable infrastructure both for supporting the user in the editing process and for program tracing and debugging. A prototype version is published on the Web; the tool can be tried by downloading the client from [3].

3.1. Architecture

In our client-server implementation of XQBE, the client provides a visual editor for the user, the server operates the XQBE-to-XQuery translation and then, if requested, executes the query by invoking one or more XQuery engines. Clients installed on machines provided with XQuery engines can also execute the queries locally.

The client and the server communicate by exchanging an internal representation of queries in an intermediate XML format, named XQBEML, that is basically an XML description of the XQBE graphs. The XQBEML representation is extended with visualization details of the query, so as to enable the graphic reconstruction of a query on the screen in the format originally defined by the user. Of course the communication protocol also supports the exchange of the result of the queries. An overall picture of the XQBE system architecture is in Figure 11.

There are many reasons why we decided to base the implementation of XQBE on a client-server architecture; the main one is that we could distribute the client as soon as it was ready with basic editing capabilities, keeping under strict control the server with the translation algorithm, which is not as stable as the client and has been evolving together with the XQuery language specification (XQuery has been in a working draft state since February 2001 and, at the time we are writing, it hasn't become a recommendation yet). We plan to distribute the server too, as soon as both the language and the translation algorithm become stable.

Another reason for decoupling the editing features and the querying capabilities is the chance that the XQBE editor is used as a thin client, possibly in a mobile scenario, on a Java-enabled PDA. In such a scenario the only required features for the client are those of drawing the query graphs and displaying the query result, without any requirement about XQuery execution capabilities or persistent XML data storage.

3.1.1. The XQBE client. The *client* has the main objective of supporting and facilitating the visual editing of the queries. The user is assisted with a strong syntactic feedback, that prevents the composition of incorrect queries. A schema-driven editing mode is available,

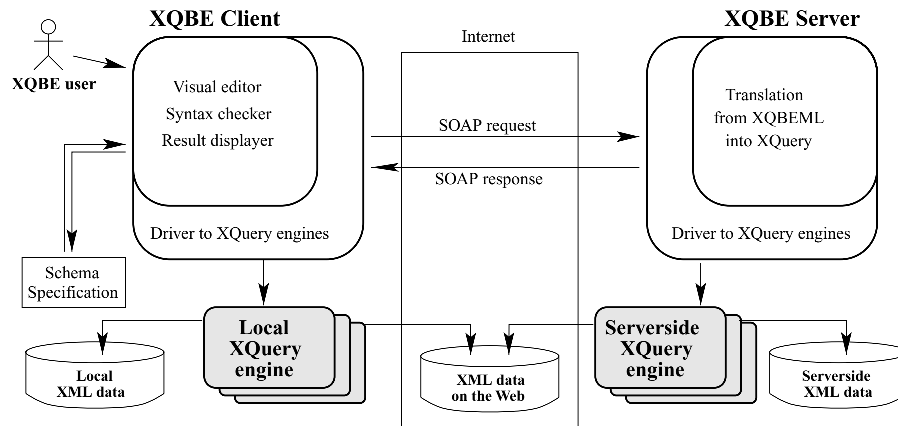

Figure 11. Overall architecture of the XQBE system.

which allows to compose the XQBE graphs with few mouse clicks to confirm and include in the query some nodes among the possible choices, incrementally and interactively offered to the user, depending on available XML Schema or DTD specifications of the target documents. These features are discussed in more detail in Section 3.3. Several snapshots of the visual interface were presented in Section 2.

During the editing process, the XQBE constructs are internally represented as XQBEML data; then they are either saved on the client or sent to the server. XQBEML maps the graphical constructs to XML tags, grouped into categories that depend on their role ad position (nodes or arcs, the side of the query they belong to, their shape); the properties of each node or arc are represented as attributes of the corresponding XML tag. XQBEML only represents the syntax (or topology) of the XQBE graphs; it contains no information about the corresponding XQuery statement. For educational and debugging purposes, the XQBEML representation of the query is visible on the client in a dedicated window.

When a query is completed and sent to the server for being processed, the application protocol allows to request either a simple translation (so that the server only returns the corresponding XQuery statement) or the query execution as well (so that the server also returns the resulting XML data). If the client only asks for the translation, the returned XQuery statement can be executed on a local XQuery engine; this mode is typical for queries targeted to local or private data, while the remote execution is typical for queries upon data available on the Web or for clients which are not provided with querying capabilities. The XQBE client packs the query represented in XQBEML into a SOAP message with several parameters that characterize the request, and sends it to the server.

3.1.2. The XQBE server. The *server* is implemented as a Web Service: when the server receives a SOAP request containing the XQBEML specification of an XQBE query, it executes the translation algorithm and then sends back a SOAP response containing the

generated XQuery statement. The part of the server that executes the translation is the core of our architecture; the translation algorithm is described and exemplified in Section 3.2. For tracing and debugging purposes, all requests are logged, together with the generated XQuery statements.

The server is also capable of executing the generated statement on several server-side XQuery engines. This optional feature is controlled by one of the parameters in the SOAP message. The query can be executed by invoking the APIs offered by several different third party query engines. The XML data produced as the result of the execution is packed in the SOAP response and sent back to the XQBE client, in addition to the XQuery statement, which is always included in the response.

Figure 12 shows that several XQuery engines can be simultaneously used to execute the same query (an idle `halloworld' query in the example). Note that a `Remote Execution' checkbox allows to control if the query is to be executed client-side or server-side (provided that the specified engines are available). Also note that the response time is displayed within the result, so that the performances of different XQuery implementations can be "synoptically" compared. None of the available XQuery implementations was correct and complete at the time we deployed the architecture, and each engine prototype had its own limitations and syntax restrictions. In order to use a common interface for them all, we implemented an `adaptation layer' that hides some differences and possibly rewrites parts of the generated XQuery statements so that they are accepted by each engine (the simplest rewriting, just to give one simple example, was the switch between the doc() and document() functions, according to the different versions of the supported XQuery specification).

3.2. Implementation of the translation algorithm

This section describes the translation algorithm that takes as input a query, composed of a set of directed acyclic graphs (DAGs) compliant with the topological constraints enforced by the syntax of XQBE, and produces as output its XQuery translation. The description considers q8 (in 2.7) to exemplify all the translation steps.

The algorithm first performs a pre-processing of the source part, so as to compute once for all the variables and the predicative terms that will be later assembled into the `for` and `where` clauses of the output query. The XQuery statement is then generated by processing the construct part with a recursive traversal; this traversal combines into `for, where`, and `return` clauses the pre-computed terms of the source part. These clauses are assembled in suitable FLWOR expressions, nested one into another according to the hierarchy of the tree structure in the construct part (and therefore of the XML data fragments to be constructed).

3.2.1. Preprocessing. The *source part* is parsed to detect those graphical configurations which map to variable definitions; these variables are used in the predicative conjunctive terms that express join conditions and selection criteria, according to the labels on the leaf nodes.

Variables are associated to some nodes in the source part; each variable is defined in terms of a path expression, corresponding to the path that reaches the node in the graph.

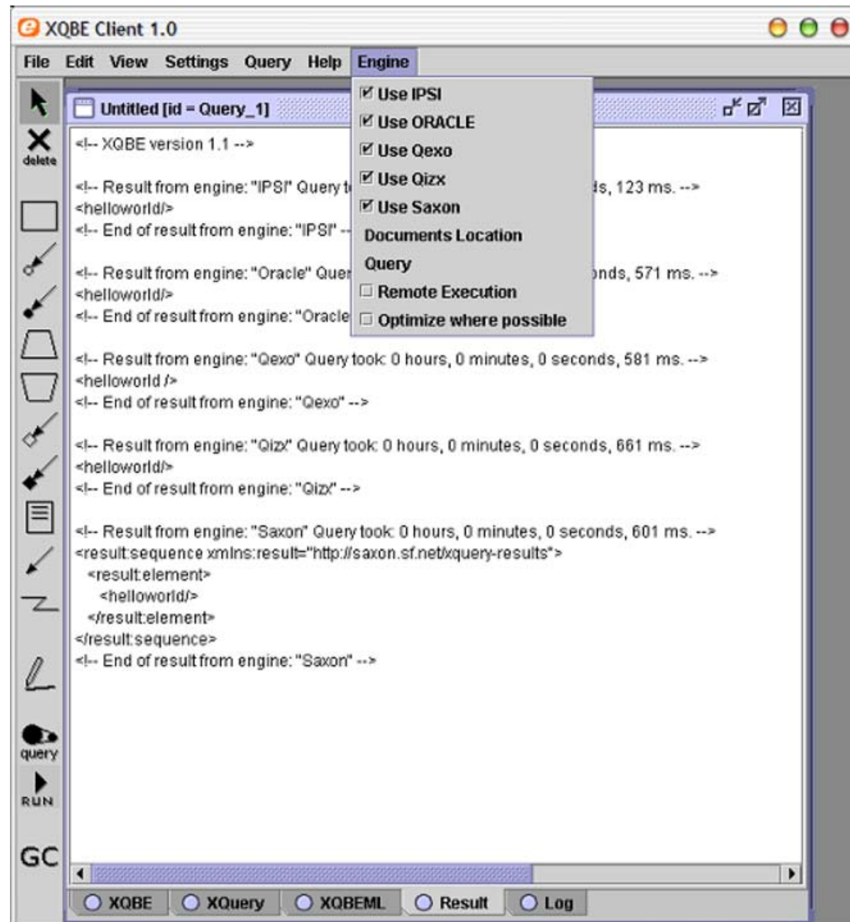

Figure 12. Query results from several XQuery engines.

These path expressions are constructed mapping each arc to the '/' step ('//' if the arc is labeled with the Kleene cross) and each node maps to its node label. Each path expression either starts from a root node (corresponding to the root element of a source document) or starts from another variable, already defined for a node above in the hierarchy. Several kinds of nodes require the instantiation of a variable:

– One variable is instantiated for each node with a *binding edge*; such variable definitions map to the `for` clauses in the XQuery statement, because binding edges control the cardinality of the nodes in the result document. In q8, the `book` and `author` nodes in the source part qualify for the instantiation of a variable, according to this criterium.

– Variables are instantiated also in correspondence with every *bifurcation*, i.e. a node with multiple outgoing arcs. These variables will help in imposing that the items in the paths branching from the node do belong to *the same* common ancestor. In q8 the book node in the construct part would qualify also according to this criterium, even if it had no binding edge; a variable for each considered book element is required in order to enforce that each considered book by "Jack" is not published by "Addison-Wesley". Without such a variable, the query would extract all Jack's books if some book by some other publisher exists in the document.
– *Leaf nodes with labels* cause the instantiation of variables to express the selection conditions they are labeled with; *Join nodes* originate predicative terms as well, taking into account the comparator associated to the node (within further predicative terms); such variable definitions are local to the `where` clauses of the XQuery statement. In q8 two such variables are instantiated, in order to denote the PCDATA content of the `publisher` and `author` elements.

All variables, paths, and predicative terms are generated with one depth-first traversal of the nodes of the source part. The traversal orderly and incrementally constructs the path expressions, generates unambiguous variable names during the descent, and builds the predicative terms for all labeled leaf nodes and all join nodes. Join predicates corresponding to a particular join node are built only when the join node has been accessed along all the incoming paths, so that all variable definitions are available for expressing the comparisons, as the variable names and path expressions are built during the descent.

Negated branches are visited in the same way, with the only restriction that the traversal does not begin until all the positive nodes have been visited. This restriction corresponds to the fact that the conjunctive where clauses are built with two levels of "nesting", with all the negated conjuncts within one term; it is thus guaranteed that all positive variables are already available for building "mixed" predicative terms in the comparisons that take place in the negated sub-clause.

Going back to q8, the preprocessing phase individuates four variables:

```
$b in doc("bib.xml")/bib/book
$f in $a/first/text()
$a in $b/author
$p in $b/publisher/text()
```

associated to the corresponding paths from the root node in the query graph. Two conditions are extracted as well, upon the values of the leaf nodes: `$f = "Jack"` and `$p = "Addison-Wesley"`. The condition about the publisher is marked as negated.

3.2.2. Processing. A depth-first traversal of the *construct part* generates a FLWOR expression[2] for each node connected by a binding edge. The for clause of this FLWOR expression defines the variable instantiated for the node on the other side of the binding.

Trapezoidal nodes are translated into trivial node constructors if their short edge is above, while they are translated into FLWOR expressions if the short edge is below, because they

require the inclusion into the result of as many new tags as the elements that are returned in correspondence with the nodes placed below in the query hierarchy, which are constructed by means of a FLWOR expression. In this second case the for clause contains as many variable definitions as the nodes that are reached by the outgoing arcs of the trapezoidal node, so that the cardinality of the generated element is that of all the combinations of the contained items (according to the interpretation of this configuration as a Cartesian product).

Whenever a bifurcation is encountered in the construct part, either the return clause of the "current" FLWOR expression is generated, by recursively visiting the branching paths, or a node constructor is generated, if the "current" node is not bound (and therefore the bifurcation is just a step in the breadth/depth projection of a node already bound above). We say that a *node* is *bound* if it has a binding edge, and that a *path* in the graph is *bound* if it contains a bound node. *Unbound* paths in the construct part which are only composed of ghost nodes are translated into path expressions that traverse the source data without including the traversed items into the result document; such path expressions are possibly enriched with filters if there are conditional nodes attached to such paths. *Bound* paths and chains of regular element nodes map to recursively nested FLWOR expressions, as they denote the inclusion of XML elements into the result and such elements have to be generated in correspondence of XML items in the source data. Such elements are matched in the for clauses of such expressions, and the corresponding output tags are constructed in the corresponding `return` clauses.

We now briefly exemplify this processing applied to the construct part of q8. The traversal starts from the trapezoidal (root) node, and generates a couple of `<list>` tags to contain the rest of the query. The recursive visit then moves to the `Jack-s-book` node, which is the vertex of a binding edge (a bound node) and thus originates a FLWOR expression. The clauses of such expressions are built as follows.

The *for* clause binds only variable `$b`, associated to book which is the other vertex of the binding edge.

The *where* clause originates from the two branches of the graph in the source part. The branch on the right contributes with the positive condition about `$f`, while that on the left with the condition about `$p`. These ingredients are combined in a two-level existential clause:

```
some $a in $b/author satisfies
  some $f in $a/first/text() satisfies
     ( $f = "Jack" and
       not( some $p in $b/publisher/text() satisfies
               ($p = "Addison Wesley" ) ) )
```

The *return* clause is built to generate the `<Jack-s-book>` tags and then a nested FLWOR expression for the `FullName` node. The decision to generate a nested FLWOR expression depends on the fact that the `FullName` node has a binding edge. The algorithm then recursively generates the nested FLWOR expression binding `$a` and extracting only a subset of the conditions, and precisely the about `$f`. The choice not to consider again

the condition on the publisher depends on the fact that `$b` has already been bound in the outer FLWOR expression, ant that binding is still visible within the scope of the FLWOR expression currently under construction. This visibility corresponds to the fact that the two binding edges of q8 are in a precise hierarchical relationship, so that the one above influences the one below. Such an influence would not hold if the edges were attached to nodes placed on parallel paths in the construct part. Accordingly, the corresponding nested FLWOR expressions would not have a common scope, as they would not be nested one into another (due to the depth-first recursive traversal strategy).

Besides the nested FLWOR expression, the algorithm also inserts a path expression that inserts into the result the corresponding `titles`, so as to conclude the "breadth" projection of the extracted `book` elements, thus completing the translation.

For the sake of readability, all the XQuery versions of queries q1 to q7 were presented as they are in the W3C Use Cases or how a human programmer would write them. Our algorithm translates the XQBE corresponding queries in a form that is equivalent, but more regular and more suited to the automatic generation.

3.3. The visual interface and its usability

This section briefly describes the features of our implementation of the XQBE visual interface.

3.3.1. The XQBE editor. The XQBE client is a Java stand-alone application that provides an editing interface similar to that of many editors for visual languages based on graphs. Users can draw their queries in windows composed of two parts, corresponding to the source and construct parts. The query graphs are built by choosing the graphical constructs from the toolbar on the left; any portion of these graphs can be cut and pasted from a query to another and the queries can be compiled to XQuery and executed with a single click.

Figure 13 demonstrates three views for query q1 (from Section 2.1). The snapshot on the left shows the "XQBE" panel, with the visual query. The snapshot above on the right shows the XQuery statement obtained from the translation server; we recall that it is not equal to the W3C version given in 2.1, but it is equivalent (the automated translation systematically exploits existential quantifications for expressing selection conditions). The snapshot below on the right shows the result of the execution of the query upon the data set of the running example (obtained by means of the default query engine, which is currently IPSI-XQ [18]).

The tool also offers the possibility to associate textual annotations to the queries. This feature mainly aims at helping unskilled users, but is also an attempt to make an XQBE query more self-explanatory, since often a filename is too short or simply unsuited for capturing the semantics and the motivation of a query. For educational purposes, in the examples distributed with the downloadable tool the annotations contain the natural language specification of the queries and a short explanation of the use of the constructs relevant for that query.

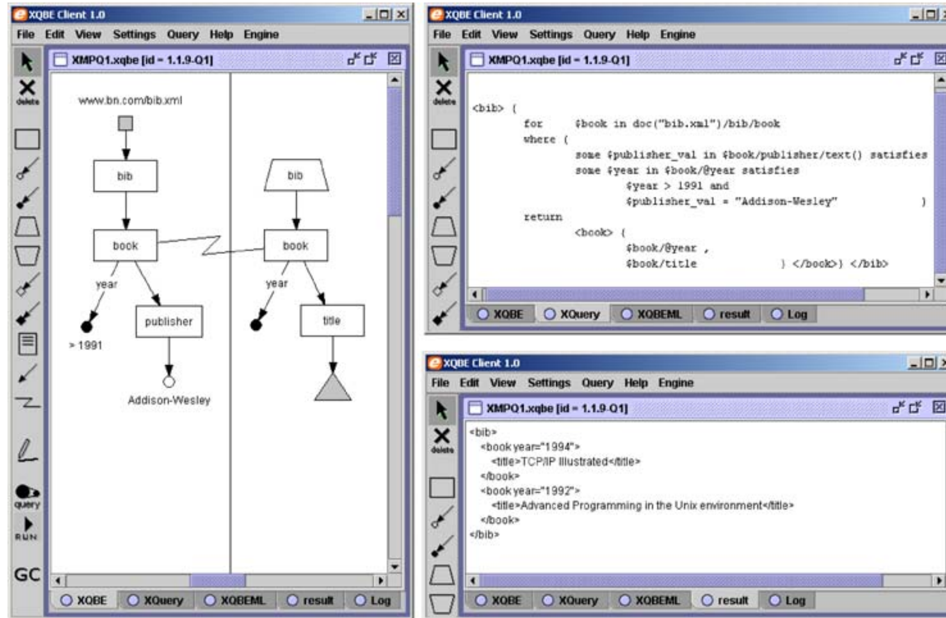

Figure 13. Editing, translation and execution of q1.

The tool assists the user in many ways during the editing process, and provides syntactic feedback to facilitate the building of correct queries. Many incorrect configurations are prevented by the tool while editing, while other feedbacks are provided at query compilation time. The syntactic feedback is not limited to detecting "topological" errors, but provides default automatic and semi-automatic corrections to typical or frequent errors, both during the query editing process and at compile time. For some "typical" errors the tool operates default automatic correction, but warns the user, who might discard the proposed modifications.

3.3.2. Schema-guided composition. The tool allows the user to build the graphs of a query by using a guided construction, with the wizard shown in Figure 14. Users can load a DTD or XML Schema definition for the target data (step 1 in the Figure), thus enabling the tool to suggest the allowed subelements of each selected item by showing its first-level expansion. The suggestions take into account the cardinality and mutual exclusion constraints. Sequences of repeated items are iteratively inserted by clicking on special "element generators". The Figure shows the first steps in loading the DTD of the running example:

```
<!ELEMENT bib (book*)>
<!ELEMENT book (title,(author+|editor+),publisher,price)>
```

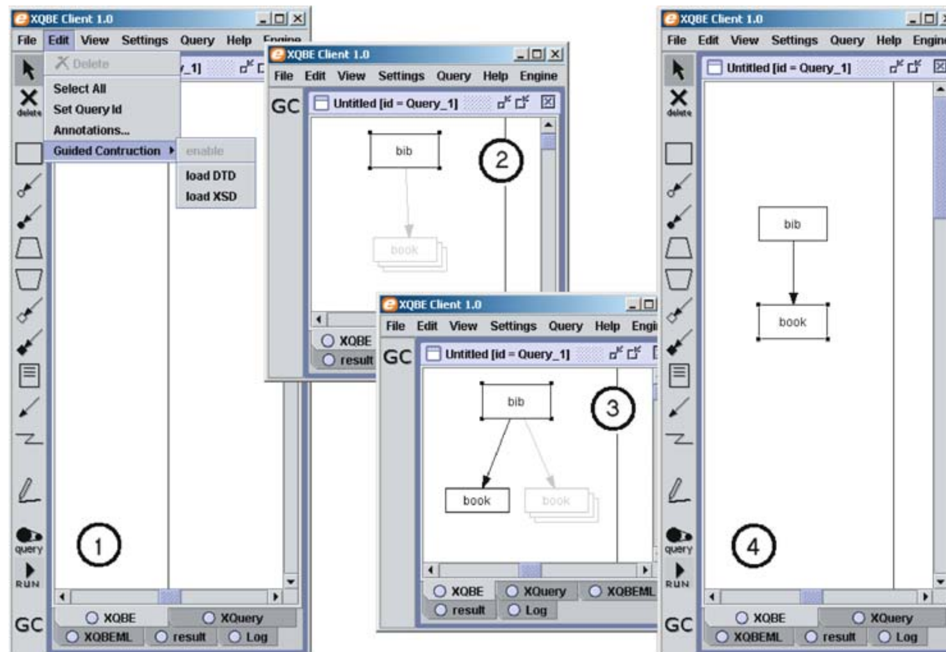

Figure 14. The schema-guided composition interface.

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

Exploiting the DTD, the user can choose the root bib element and expand it to its sub-nodes. The wizard (step 2 in the Figure) only shows one book element generator as descendant of bib, with a new icon that suggests the containment of multiple elements. Indeed, according to the DTD, the only legal subelements are book elements. Also, the node is shown in grey and in a smaller size, so as to stress the fact that it is just a suggestion to the users, not yet a node included in the query.

Users can "confirm" the nodes they want to include (which become black) with a single click, and then recursively expand them in turn, thus incrementally constructing a tree with a navigational "explorative" paradigm. Step 3 in Figure 14 shows what happens clicking on

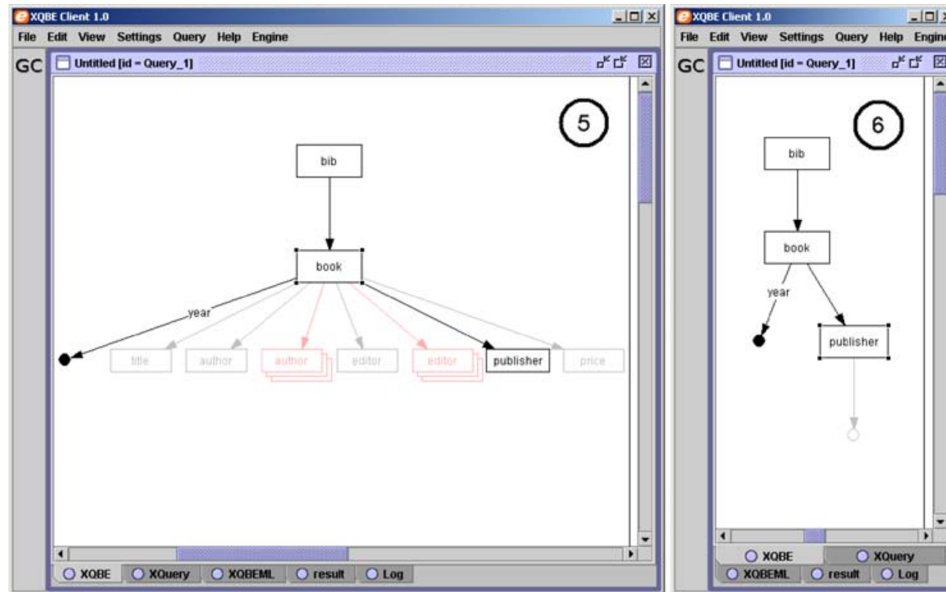

Figure 15. The schema-guided composition interface (continued).

the generator icon: a black book icon is generated, and the grey one remains for generating other book nodes. When the user disables the wizard (as in step 4) all the unconfirmed nodes disappear, while the selected (black) ones become regular components of the query.

Figure 15 shows how the wizard expands a `book` element into the tree of its components (step 5, where only the `year` attribute and the `publisher` have been confirmed and are therefore in black). Step 6 shows that, with a further descent, only the PCDATA node of the publisher is shown in grey (there is no further legal content for `publishers`): the skeleton of the source part of q1 has been constructed with six mouse clicks and is now ready for being labeled with the selection conditions.

The construction wizard also prevents the user from confirming an element if it is in mutual exclusion with a confirmed subelement, and exploits colors to highlight these constraints; the forbidden elements are displayed in red, clicking on them has no effect, and they become selectable again only when the alternatives are deselected. According to the DTD of the running example, authors are mutually exclusive with editors, and Figure 16 shows that choosing an author disables the editors (which are still displayed, but in red), while choosing an editor disables the authors.[3]

The description of the schema-aware guided construction has focused on a DTD specification, because the equivalent XML Schema would be too verbose to fit these pages. Indeed, the user would not notice the difference, as the wizard is strictly based on the XQBE data model, that only distinguishes between tags, attributes, and text. XML Schema specifications (and also DTDs in their full capabilities) are more expressive than the constraints

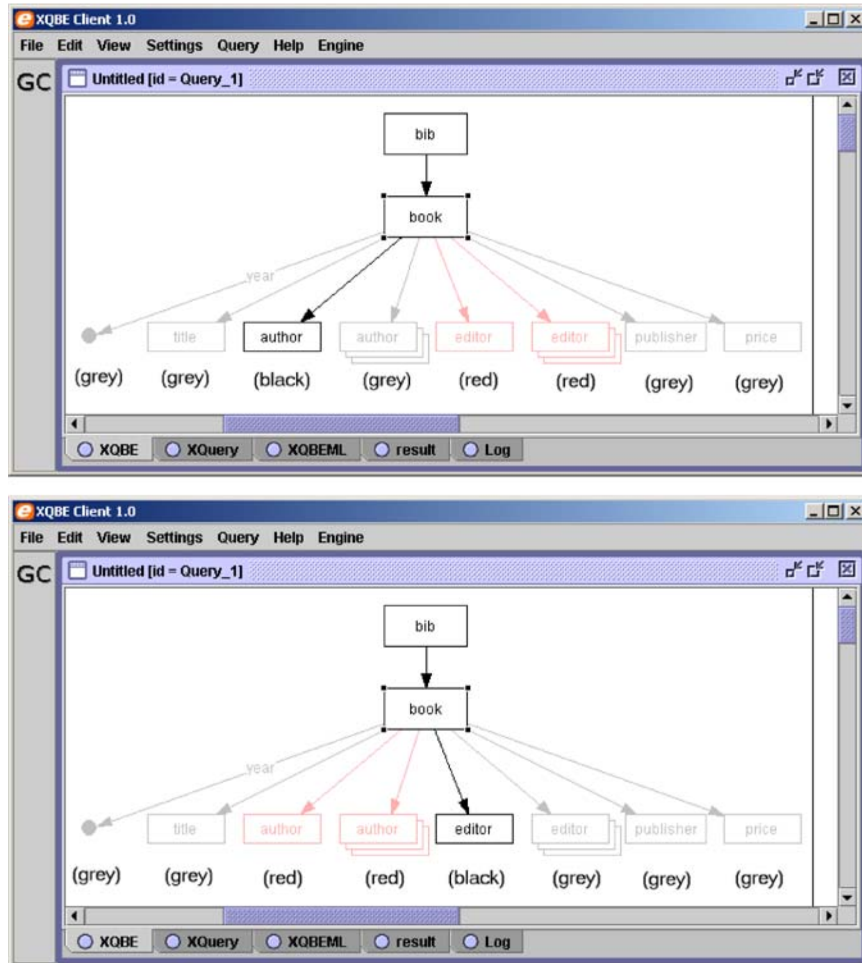

Figure 16. Mutual exclusion in the guided composition.

representable with the composition wizard; indeed, `.dtd` and `.xsd` files are parsed in order to extract only those constraints that are applicable to XQBE. The data model underlying XML Schema specifications, in particular, is far richer than that of XQBE, which does non support data types, namespaces, substitution groups, and so on. Once again, according to the XQBE "philosophy", a too sophisticated interface has been discarded in favor of a wizard capable of giving intuitive support for querying data sets whose structure is only partially known or unknown at all.

Last, it is worth noting that the schema is exploited only for giving suggestion, not for validating purposes: a user can deliberately draw a structure not compliant with a particular schema, say, for finding those documents that violate that schema.

3.4. Usability test

The usability of the language and the tool is the primary characteristic of XQBE. The intended final user of XQBE is any user who is at least partially aware of the basics of the XML data model and has a query to be performed on some XML data whose schema is somehow well known, either because its XSD/DTD specification is available or because the user is confident with it. We performed an extensive testing activity in order to obtain the desired level of usability. XQBE was not born with the current syntax and semantics; instead, it evolved through several intermediate steps up to the present version. Every version of the language was tested mainly by graduate and undergraduate students of the database courses at Politecnico di Milano. We taught (and still teach) the basics of the XML data model and the syntax and semantics of XQBE in a two hour introductory lecture; XQuery in its full complexity comes in our courses only after this introduction, and we evaluate the students' ability to formulate simple and not-too-simple queries and transformation both on paper and using the tool. The evolution of the language and the prototype took into account several kinds of feedbacks which e got from users. As an example, the first version of XQBE, which was published in [4], had two different kinds of binding edges, separately addressing the problem of transferring values from the source to the construct part and the problem of enforcing the cardinality of the sets of XML fragments included in the result. In the first revision we decided that one king of binding was sufficient and decided to disallow multiple binding edges for a node in the construct part - two characteristics which were perceived as confusing. Several cycles of revision led us to establish the final XQBE version [5], accurately trading between the expressive power and the intuitiveness and ease of use of the language. The same type of analysis was applied also to the tool: and the version presented in this paper is also the result of several revisions.

4. Related work

Since the early days of XML, several textual query languages were proposed and analyzed by the database community [16,19], far before the proposal of XQuery [28]. XQBE, in turn, comes after a long stream [7] of research on graph-based logical languages, started many years ago with QBE [29], a language based on the visual representation of tables and conditions.

A relationally complete visual query language that supports recursion (specifically designed for relational data) is QBD* [1]. QBD* is characterized by a uniform graphical interface for both schema specification and query formulation, based on the use of an Entity-Relationship oriented data model. The main idea of the system is to provide the users with a large set of graphical primitives, in order to friendly extract the required information from the database schema and deal uniformly with the same graphical environment during all the interaction with the database, without textual intermediate. According to this approach, the visual data model of XQBE is isomorphic to the target XML data, and the availability of a schema specification is exploited for "navigating" along the hierarchies at query formulation time.

The first graph-based query languages with recursion were G [14] and G+ [15]; they are targeted to data sets represented as graphs and are general enough to query also relational data, as long as it is represented as a graph. Graphlog [13] is a direct descendant of G+. A uniform notation for object databases where nodes represent objects and edges represent relationships was used in Good [24]. A Good-like notation was used by G-Log [25], a logic-based graphical language that allows to represent and query complex objects by means of directed labeled graphs. An evolution of this language, WG-Log [12], was built to query internet pages and semi-structured data adding to G-Log some hypermedia features. A direct descendent of WG-Log is XML-GL [11], an early and self-standing visual query language for XML, designed far before XQuery.

XQBE can be considered a successor to XML-GL, however with several new features. Due to the specificity of XQuery, new constructs have been introduced from scratch and some constructs of XML-GL have been revised. The semantics has significantly changed in order to facilitate the translation into XQuery. Thanks to these extensions, several queries not expressible with XML-GL are very easily expressible with XQBE (and with XQuery). For example, XML-GL does not provide the capability to specify conditions in the construct part, nor that of projecting "in depth" the extracted fragments without imposing at the same time an existential condition in the left side of the query.

QSByE (Query Semi-structured data By Example [17]) is a graphical interface that represents data as nested tables and extends the QBE paradigm to deal with semi-structured data. MiroWeb Tool [2] uses a visual paradigm based on trees that implements XML-QL. QBEN is a graphical interface to query data according to the nested relational model; the users specify their queries with the operations of the nested relational algebra [20]. Equix [9] is a form-based query language for XML repositories, based on a tree-like representation of the documents, automatically built from their DTDs. Intra-document relationships cannot be visually rendered. Equix has limited restructuring capabilities: the only restructuring primitive is the introduction of new nodes, containing aggregation values (sum, count, max, ...). In [10] a new syntax for Equix is proposed, more user-friendly but limited to searching the Web. BBQ [22] (Blended Browsing and Querying) is a graphical user interface proposed for XMAS [21], a query language for XML-based mediator systems (a simplification of XML-QL). In BBQ XML elements and attributes are shown in a directory-like tree and the users specify possible conditions and relationships (as joins) among elements. The expressive power of BBQ is higher w.r.t. Equix, but restructuring capabilities are limited and aggregations are not supported.

PESTO [6] (Portable Explorer of STructured Objects) is an integrated user interface that supports browsing and querying of object databases; PESTO allows users to navigate in a hypertext-like fashion, following the relationships that exist among objects. In addition, it allows users to formulate object queries through a unique, integrated query paradigm that presents querying as a natural extension of browsing. PESTO includes support for basic query operations (such as simple selections, value based joins, universal quantification, negation, and complex predicates). VQBD [8] address the objective to explore an XML document of unknown structure.

XQForms [26] is a generator of Web-based query forms and reports for XML data. XQForms takes as input the XML Schema, a declarative specification of the logic of the

query and a set of template libraries. The usage of these three different inputs allow a clear separation between data to be queried, query logic and presentation of the results.

QURSED [23] allows the development of web-based query forms and reports (QFRs) for XML data. QURSED produces XQuery-compliant queries. The QURSED Editor inputs the XML Schema describing the structure of XML data and an HTML query form page (that provides the visual part of the form page). The editor displays the XML Schema and the HTML pages to the developer, who uses them to visually build the query set specification and the query/visual association (that indicates how each parameter is associated to HTML form). Then a compiler generates Java Server Pages, which control the interaction with the end user.

5. Conclusions and future work

In this paper, we presented XQBE, a graphical query language that offers a visual interface to query XML documents. This contribution may stimulate academic and industrial research in a field that is fundamental to the success of XQuery for a wider audience.

We have also presented a prototype implementation of XQBE based on a client-server architecture; it provides a visual editor for our proposed interface and translates graphical queries into XQuery, so that it can be used in conjunction with any XQuery-compliant query engine. The prototype can benefit of available schema specifications of the source documents; such information can be exploited to guide the users in the composition of their query, especially if they are not familiar with the data set they have to manage.

There are several potential opportunities for future work. The main undergoing extension of is to allow the generation of XSLT translations for XQBE queries.

Another opportunity is that of specializing XQBE and its implementation with constructs, primitives and capabilities specific to some applicative domain, to provide a simple and visual language for "information extraction" tasks. We are currently planning an attempt in this direction, in cooperation with a research group that develops a system for fast and efficient access to digital libraries with semi-structured data.

From a usability viewpoint, we are designing an integrated environment to support both XQuery and XQBE, where users can use the graphic tool to produce textual queries and/or to produce the XQBE view of a given XQuery statement.

Acknowledgments

We wish to thank Stefano Ceri, Letizia Tanca, and Sara Comai for the interesting discussions and useful suggestions. We are grateful to Enrico Augurusa, Alessandro Raffio, Massimo Sarchi, and Luca Lulani for their fundamental help with the implementation; we also deem valuable the contribution of Marco Sartor, Simone Tognetti, Alessandro Vacca, and Paolo Tomasi

Notes

1. The full schema and instance of `review.xml` are available in [27].
2. We recall that one of the basic constructs of XQuery is the so-called FLWOR expression, acronym of For-Let-Where-Orderby-Return, the names of the clauses of the construct.
3. In the figure the nodes are labeled with the color they are displayed in (in case of the grey-scale print).

References

[1] M. Angelaccio, T. Catarci, and G. Santucci, “QBD*: A graphical query language with recursion,” *IEEE Transactions on Software Engineering* 16(10), 1990, 1150–1163.

[2] L. Bouganim, T. Chan-Sine-Ying, T.-T. Dang-Ngoc, J. L. Darroux, G. Gardarin, and F. Sha, “Miro web: Integrating multiple data sources through semistructured data types,” in *Proc. of the 25th Int. Conf. on Very, Large Data, Bases (VLDB’99),* Edinburgh, Scotland, UK, 1999, pp. 750–753.

[3] D. Braga, and A. Campi, 2003, "XQBE, W,eb Site". http://dbgroup.elet.polimi.it/XQBE.

[4] D. Braga, A. Campi, and S. Ceri, “A graphical environment to query XML data with XQuery,” in *Proc. of the ACM-SAC 2003,* Melbourne, Florida, USA, 2003

[5] D. Braga, A. Campi, and S. Ceri, 2005, “XQBE (XQuery by example): A visual interface to the standard XML query language,” *ACM Transactions on Database Systems (TODS)*. To appear in June 2005.

[6] M. Carey, L. Haas, V. Maganty, and J. Williams, “PESTO: An integrated querybrowser for object databases”. in D. McLeod, R. Sacks-Davis, and H. Schek (eds.), in *Proc. of the 22nd Int. Conf. on Very, Large Data, Bases (VLDB’96)*, 1996, pp. 203–214.

[7] T. Catarci, M. F. Costabile, S. Levialdi, and C. Batini, “Visual query systems: Analysis and comparison,” *ACM TODS—Transactions on Database Systems* 8(2), 1997, 215–260.

[8] S. Chawathe, T. Baby, and J. Yeo, “VQBD: Exploring semistructured data (demonstration description),” in *Proc. of the ACM SIGMOD*, 2001, p. 603.

[9] S. Cohen, Y. Kanza, Y. A. Kogan, W. Nutt, Y. Sagiv, and A. Serebrenik, “EquiX easy querying in XML databases,” in *WebDB (Informal, Proceedings)*, 1999, pp. 43–48.

[10] S. Cohen, Y. Kanza, Y. A. Kogan, W. Nutt, Y. Sagiv, and A. Serebrenik, “Combining the power of searching and querying,” in *5th Int. Conf. on Cooperative, Information Systems*, 2000.

[11] S. Comai, E. Damiani, and P. Fraternali, “Computing graphical queries over XML data,” *ACM TOIS* 19(4), 2001, 371–430.

[12] S. Comai, E. Damiani, R. Posenato, and L. Tanca, “A schema based approach to modeling and querying WWW data,” in *FQAS’98*.1998, pp. 110–125.

[13] M. P. Consens, and A. O. Mendelzon, “The G+/GraphLog visual query system,” in *Proc. of the 1990 ACM SIGMOD,* Atlantic, City, NJ, May 23–25, 1990, p. 388.

[14] I. F. Cruz, A. O. Mendelzon, and P. T. Wood, “A graphical query language supporting recursion,” in *Proc. of the ACM SIGMOD* 1987, pp. 323–330.

[15] I. F. Cruz, A. O. Mendelzon, and P. T. Wood, “G+: Recursive queries without recursion,” in *2nd Int. Conf. on Expert, Database Systems*, 1988, pp. 355–368.

[16] M. Fernandez, J. Siméon, P. Wadler, S. Cluet, A. Deutsch, D. Florescu, A. Levy, D. Maier, J. McHugh, J. Robie, D. Suciu, and J. Widom, 1999, “XML query languages: Experiences and exemplars,” http://www-db.research.belllabs.com/user/simeon/xquery.ps.

[17] I. M. R. E. Filha, A. H. F. Laender, and A. S. da Silva, “Querying semistructured data by example: The, Qsbye interface,” in *Workshop on Information, Integration on the Web*, 2001, pp. 156–163.

[18] Fraunhofer Gesellschaft IPSI: 2003, “IPSI-XQ — The, XQuery Demonstrator”. http://ipsi.fhg.de/oasys/projects/ipsi-xq/index_e.html

[19] Z. G. Ives and Y. Lu, “XML query languages in practice: An evaluation,” in *Proc. of WAIM’00*, 2000, pp. 29–40.

[20] G. Jaeschke and H. J. Schek, “Remarks on the algebra on non first normal form relations,” in *Proc. of 1st ACM SIGACT-SIGMOD, Symposium on the Principles of Database, Systems*, 1982, pp. 124–138.

[21] B. Ludaescher, Y. Papakonstantinou, P. Velikhov, and V. Vianu, "View, Definition and DTD, inference for XML," in *Proc. Post-IDCT, Workshop*, 1999.
[22] K. Munroe and Y. Papakonstantinou, "BBQ: A visual interface for browsing and querying XML," in *Proc. of the 5th Working Conference on Visual Database Systems*, 2000, pp. 277–296.
[23] Y. Papakonstantinou, M. Petropoulos, and V. Vassalos, "QURSED: querying and reporting semistructured data," in *Proc. of the ACM SIGMOD*, 2002.
[24] J. Paredaens, J. V. den Bussche, M. Andries, M. Gemis, M. Gyssens, I. Thyssens, D. V. Gucht, V. Sarathy, and L. V. Saxton, "An overview of GOOD," *SIGMOD Record* 21(1), 1992, 25–31.
[25] J. Paredaens, P. Peelman, and L. Tanca, "G-Log a declarative graph-based language," *IEEE, T,rans. on Knowledge and Data Eng*, 1995.
[26] M. Petropoulos, V. Vassalos, and Y. Papakonstantinou, "XML Query Forms (XQForms): Declarative, Specification of XML Query Interfaces," in *Proc. of the 10th WWW, Conference*, 2001.
[27] W3C: 2004a, "XML, Query Use, Cases," http://www.w3.org/TR/xmlquery-use-cases
[28] W3C: 2004b, "XQuery: An, XML, query language," http://www.w3.org/XML/Query
[29] M. M. Zloof, "Query-by-Example: A data base language," *IBM, Systems Journal* 16(4), 1977, 324–343.