A Graphical Environment to Query XML Data with XQuery

Daniele Braga
braga@elet.polimi.it

Alessandro Campi
campi@elet.polimi.it

Dipartimento di Elettronica e Informazione, Politecnico di Milano
Piazza Leonardo da Vinci 32, 20133 Milano, Italy

Abstract

XQuery, the standard query language for XML, is increasingly popular among computer scientists with a SQL background, since queries in XQuery and SQL require comparable skills to be formulated. However, the number of these experts is limited, and the availability of easier XQuery "dialects" could be extremely valuable.

With this motivation in mind, we designed XQBE, a dialect of XQuery inspired by the QBE language (Query by Example). QBE, initially proposed as an alternative to SQL, has then become popular as the user-friendly query language supported by MS Access.

XQBE starts from hierarchical structures, coherent with the hierarchical nature of XML, and uses one or more structures to denote the input documents, and one structure to denote the XML document produced in output. These structures are annotated to express selection predicates; explicit bindings connecting the nodes of these structures visualize the input/output mappings.

1 Introduction

The diffusion of XML in most 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 [27] 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.

1.1 Motivation and design principles

The W3C (World Wide Web Consortium) provides two textual languages to express XML document transformations and to query XML data, XSLT [23] and XQuery [25][1]. These languages, however, are far too complicated for occasional or unskilled users, who might only be aware of the basics of the XML data model, or simply conscious of the schema of the documents they have to manage. Nevertheless, this basic knowledge should be enough to allow any user to express queries and transformations with the basics of a suitably simple (maybe ad hoc) query language.

XQBE was designed with the objectives of being intuitive (according to the aforementioned principles) and of being 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 (except some of its core functions, such as `position()`, and operators, such as `instance of`). XQBE allows for arbitrarily deep nesting of XQuery FLWOR expressions, 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, which is Turing-complete (a proof can be found in [17]). 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

[1]For the sake of brevity, we assume the reader is familiar with XQuery. If that was not the case, a good introduction is the W3C Use Case [24]. In any case, we trust that at least our visual queries will be readable and comprehensible even to those without a specific XML background.

```
<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)

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

This paper first presents XQBE by means of several examples taken from the W3C "XML Query Use Cases" [24], then discusses how XQBE is translated into a subset of XQuery, whose definition is given by means of a BNF grammar. The paper is completed by the description of our implementation of XQBE and concludes with a section about previous related research work.

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 in the W3C *XMP* XML Query Use Cases ([24]), based on the XML fragment of Figure 1.

2.1 Simple queries

The first query, q1 (named Q1 in [24], section 1.1.9) reads "*List books published by Addison-Wesley after 1991, including their year and title*":

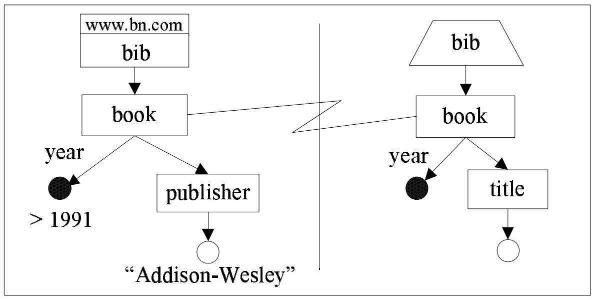

Figure 2. A query with match conditions (q1)

```
<bib>
 { for $b in document("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>
```

Its XQBE version is depicted in Figure 2. A query always has a vertical line in the middle, that separates the *source* part (the one on the left) from the *construct* part (that on the right, so that the query has a "natural" reading order from left to right). Both parts contain labelled 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 bindings across the vertical line and connect the nodes of the source part to the nodes that will take their place in the output document.

All the XML *elements* in the target document are depicted as labelled rectangles, their *attributes* are depicted as black circles (with the attribute *name* on the arc between the rectangle and the circle), and their PCDATA content is always depicted as an empty circle. The black and empty circles are named *value* nodes. Value nodes may be labelled, so as to express conditions on the values they represent.

In Figure 2 we extract 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 `bib` node in the source part, nodes can be labelled with URLs 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 `book`s are retained). The binding

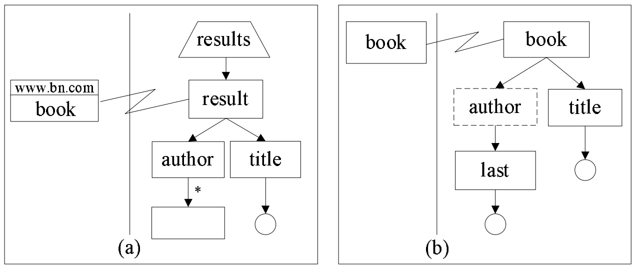

Figure 3. Renaming, breadth/depth projection (q2, q3)

edge between the `book` nodes states that the query result shall contain *as many* `book` elements *as* those matched in the source part. The trapezoidal `bib` node above the `book` node means that all the generated `book`s are to be contained into a single `bib` element. This node represents a newly generated element, and *new elements* are always depicted as trapezia in XQBE. Trapezia can be depicted with their short edge on the upper side or on the bottom side. These two configurations impose different cardinality constraints on the newly generated elements[2].

2.2 Element projection and renaming

Query Q3 in [24] reads "*For each book in the bibliography, list the title and authors, grouped inside a* `result` *element*", and translates to:

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

In this query there are no conditions upon values, but only a "projection" with "renaming" of all the `book` elements (q2 in Figure 3a): the binding edge between the `book` element and the `result` element causes the construction of a `re-sult` element for each `book`. The `author` and `title` elements below `result` will extract the corresponding subelements of `book`, thus projecting the `book` element, which is bound to `result`. The unlabelled node below `author` states that all its subelements are to be included in the generated result, while the asterisk on the incoming arc extends the inclusion to any level of depth. This configuration is therefore used to synthetically include entire fragments in the result (the asterisk reminds of the Kleene star to express the closure of the containment relationship).

[2]When the short edge of the trapezoid 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 a single tag. When the short edge is below, each such item is contained into a different instance of the newly generated element.

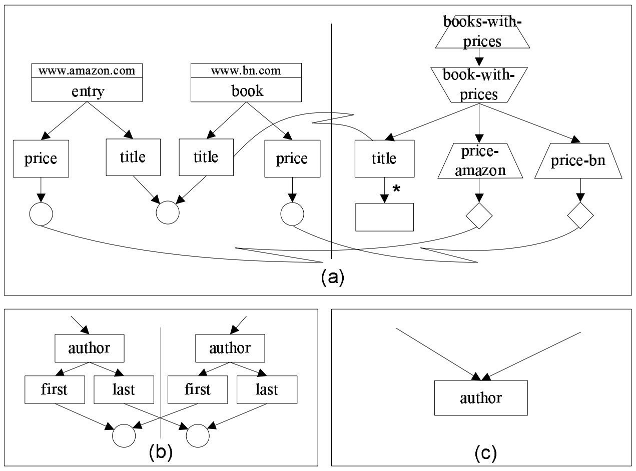

Figure 4. Join between two documents (q4)

The previous query projects 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). As an example consider a query that reads "*For each book list only the title and the surnames of the authors (maintaining the books in the order of the original document)*". It maps to the following XQuery statement:

```
for $b in //book
return <book> { $b/title }
              { $b/author/last } </book>
```

In this case the `author` elements are pruned from the generated result, and are thus represented in XQBE depicting the `author` node in the construct part with dashed lines (see q3 in Figure 3b); `last` elements are directly inserted into the `book` elements. Before stepping on to more complicated configurations, we show in Figure 5 all the constructs of XQBE. The figure lists all the graphic components (above), while the left and right parts (below) show some common remarkable configurations, together with their interpretations, when they are depicted in the source and construct part respectively.

2.3 Join between two documents

Query q4 (Q5 in [24]) 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*":

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

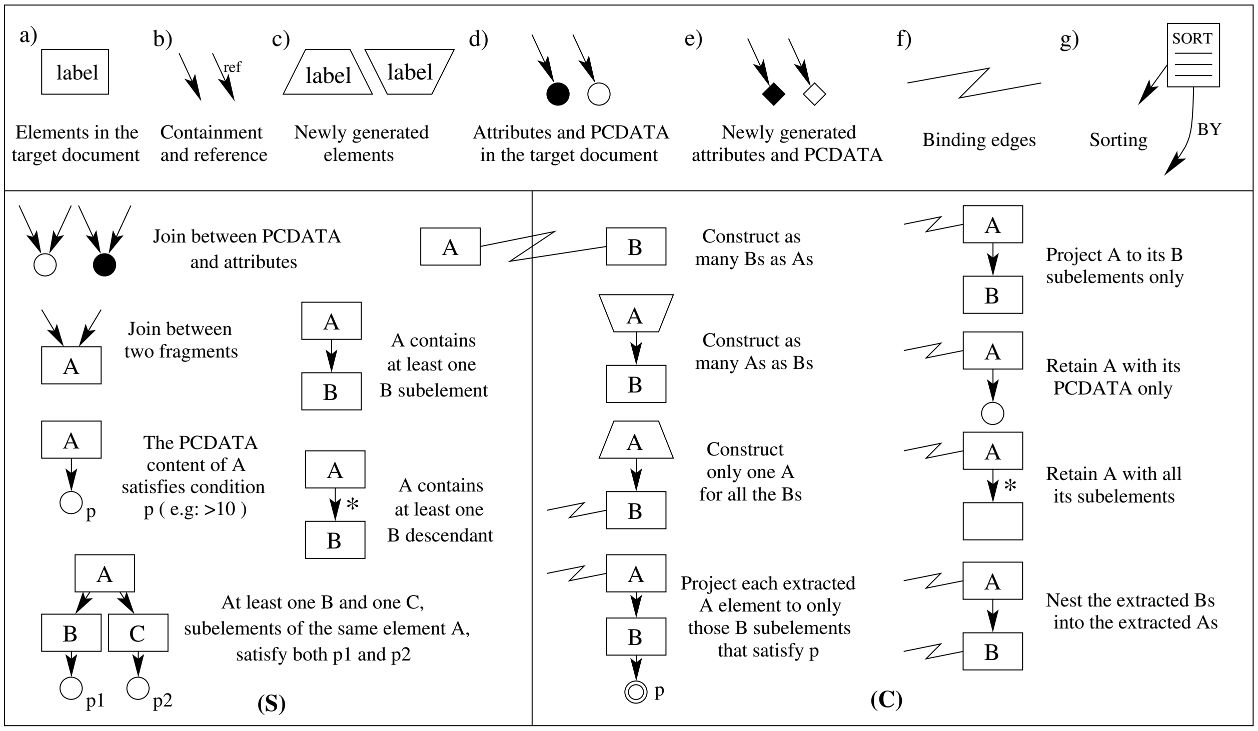
a)
label
Elements in the target document
b)
ref
Containment and reference
c)
label
label
Newly generated elements
d)
Attributes and PCDATA in the target document
e)
Newly generated attributes and PCDATA
f)
Binding edges
g)
SORT
Sorting
BY
Join between PCDATA and attributes
Join between two fragments
A contains at least one B subelement
The PCDATA content of A satisfies condition p (e.g: >10)
A contains at least one B descendant
At least one B and one C, subelements of the same element A, satisfy both p1 and p2
p
p1
p2
(S)
Construct as many Bs as As
Construct as many As as Bs
Construct only one A for all the Bs
Project each extracted A element to only those B subelements that satisfy p
(C)
Project A to its B subelements only
Retain A with its PCDATA only
Retain A with all its subelements
Nest the extracted Bs into the extracted As

Figure 5. The complete set of graphical constructs and some characteristic configuration in the source (S) and construct (C) parts

```
      { $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 4a the equality between the values is expressed by means of the confluence into a *single* value node, that represents the PCDATA content of *both* the `title` elements. More generally, the language allows to specify conditions by labelling such "confluence" nodes with binary predicates ($<, <=, ! =, ...$), while equality is the default if unspecified.

The `price-amazon` and `price-bn` elements are depicted as trapezia because they are new nodes. Their PCDATA content cannot be automatically determined, so the user has to explicitly bind their value nodes to appropriate nodes in the match part. Note that these new PCDATA contents are represented as rhombuses, to stress the fact that they are built from scratch.

The join in the previous query 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 4b, while the compact notation supported by XQBE is that of Figure 4c (where one node stays for eight). 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. This shorthand allows to draw q4 also as in Figure 11.

2.4 Document restructuring

XQBE allows to express many kinds of document transformations by means of the constructs that have been illustrated so far. As an example, we show now how to synthetically express the flattening of hierarchical data structures. Query q5 (Q2 in [24]) reads "*Create a flat list of all the title-author pairs, with each pair enclosed in a* `result` *element*", and translates to the following XQuery statement:

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

Its visual version is depicted in Figure 6. 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

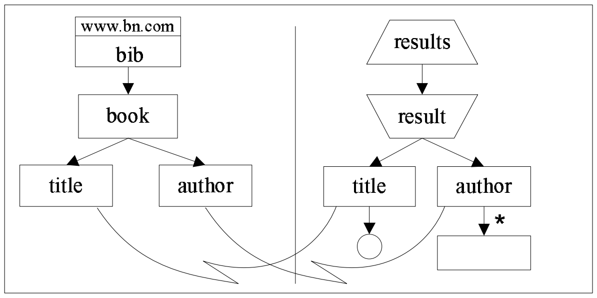

Figure 6. Document restructuring (q5)

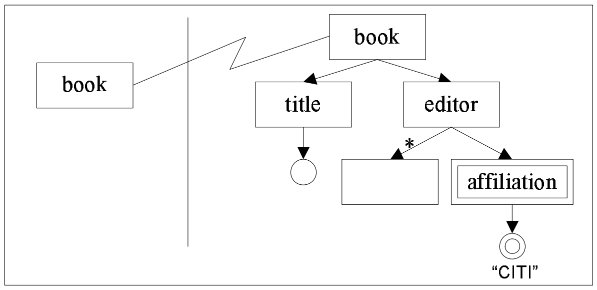

Figure 7. Filters in the construct part (q6)

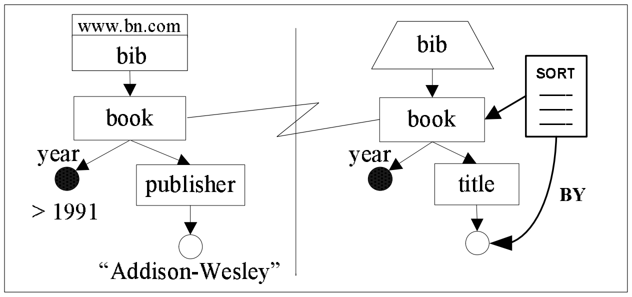

Figure 8. Sorting (q7)

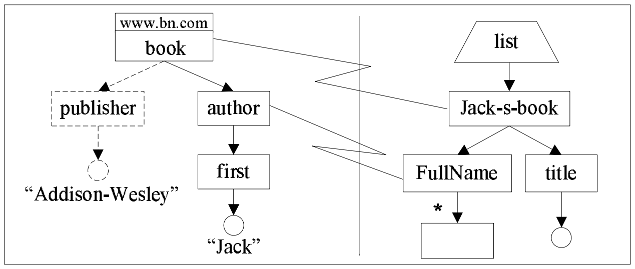

Figure 9. Nesting and negation (q8)

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 `title`s and `author`s that are contained into the same `book` are considered. Also note that the correspondence between the `title`s and the `author`s 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*".

```
for $b in //book
return <book> {$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, depicted in double lines (see q6 in Figure 7). In general, a double lined subtree restricts only the element in which it is rooted.

2.6 Sorting

Consider the statement q7 (an extension of query 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 [24]):

```
<bib>
 { for $b in document("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 `sortby` clause. Accordingly, in the graphical representation we just need to add a `SORT` node (depicted in bold in Figure 8).

2.7 Nesting and negation

As a last example consider the following XQuery statement (the XQBE version is in Figure 9):

```
<list>
 { for $b in document("www.bn.com/bib.xml")//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>
```

which translates a query that reads "*List all the 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*". This query contains *negated nodes* (those depicted in 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 dashed nodes represent a negated condition in the source part, and represent elements that are not to be retained in the result when depicted in the construct part. The meaning is clearly different, but in both cases matching XML items *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.

3 Semantics of XQBE

This section formally defines the subset of XQuery which can be generated by the XQBE interface. We describe this subset by means of an EBNF grammar.

3.1 EBNF specification of the output language

In the grammar of Figure 10 terminals are enclosed in single apexes, nonterminals in angle braces. Some semantic constraints are described within square brackets.

A query always translates to *one* XQuery FLWR expression, possibly contained into an arbitrary number of node constructors, according to production {1}. These node constructors would translate a corresponding sequence of trapezia (with the short edge above), when such a configuration is the root of a tree in the construct part.

The *for* clause of the outmost FLWR expression contains a list of variable bindings {3}, each of which corresponds to a binding edge. Variables are bound to a restricted class of path expressions (<`unfiltered_expr`> in {4} and {16}), because we chose to express all the matching conditions in the where clause.

Where clauses have a fixed structure {5-13}. We extensively use the existential quantifier `some...satisfies`, which vividly and unambiguously expresses the "existential" nature of the conditions in the source part. In other words, whenever a node is depicted with a subelement annotated with selection predicates, the semantics of this configuration is that the node *must* contain *at least one* subelement that matches the selection condition. As an example consider again query q8, where the books are extracted if *at least one* of their authors is named "Jack". The clause is in a typical conjunctive normal form (with all the quantifiers first) and has at most one nested negated clause, which in turn is in the same normal form. The atomic terms are comparisons of values and arithmetic expressions {9-11}, {18}. This rigid constraint helps both the user's intuition and the automated translation: all the predicates with which a graph is annotated are visible "at the same time" and the most intuitive requirement is that they hold "all together"; moreover the predicates can be easily collected with a recursive visit of the graphs and assembled in flat clauses. This "synoptical evidence" is also the motivation for our decision not to support disjunction.

The *return* clause always constructs a new element {14}, which contains a collection of "projecting expressions" {15}. This collection was named like this to recall that the return clause projects "in breadth and depth" the elements extracted by the for clause; such expressions can be either path expressions or nested FLWR expressions. The path expressions in the return clause may contain filters, to translate filtering nodes (described in Section 2.5).

3.2 Translation principles

The algorithm takes as input a query, composed of a set of graphs compliant with the syntax of XQBE, and produces as output its XQuery translation, a sentence of the XQuery subrange defined in Section 3.

The translation starts with a preprocessing of the source part, so as to compute once for all the variable bindings and the predicative terms to be put into the clauses of the output query. The query is then constructed by processing the construct part with a recursive visit that takes advantage of the pre-computed terms.

3.2.1 Preprocessing

The graphs in the *source part* are parsed in order to detect those graphical configurations, which are mapped to variable definitions. These variables are used to construct the predicative terms that express the selection criteria in the labels of the leaf nodes.

Each node that is reached by a *binding edge* causes the instantiation of a variable. This variable is associated to

```
 {1} <query>           ::= <flwr_expr>   |   <start_tag> '{' <query> '}' <end_tag>                [matching tags]
 {2} <flwr_expr>       ::= <for_clause> <where_clause> <return_clause>
 {3} <for_clause>      ::= 'for' <var_binding> ( ',' <var_binding> )*
 {4} <var_binding>     ::= <variable> 'in' <unfiltered_expr>
 {5} <where_clause>    ::= 'where' <ex_quantifier>* <quantified>
 {6} <ex_quantifier>   ::= 'some' <var_binding> 'satisfies'
 {7} <quantified>      ::= '(' <atom_list> ')'   |   '(' <atom_list> 'and' <neg_clause> ')'
 {8} <atom_list>       ::= <atom> ( 'and' <atom> )*
 {9} <atom>            ::= <pred_term>   |   'exists(' <filtered_expr> ')'
{10} <pred_term>       ::= <expression> <comparator> <expression>
{11} <expression>      ::= <constant_value>  |  <variable>  |  <computed_value>     [already bound variables only]
{12} <comparator>      ::= 'eq'  |  'lt'  |  'gt'  |  'le'  |  'ge'  |  'ne'
{13} <neg_clause>      ::= '( not (' <ex_quantifier>* '(' <atom_list> ')))'
{14} <return_clause>   ::= 'return' <start_tag> <projection_list> <end_tag>                   [matching tags]
{15} <projection_list> ::= <start_tag> <projection_list> <end_tag> <projection_list>  |          [matching tags]
                           '{' <filtered_expr> '}' <projection_list>  |
                           '{' <flwr_expr> '}' <projection_list>  |
{16} <unfiltered_expr> ::= XPath expression without filtering (predicative) steps
{17} <filtered_expr>   ::= XPath expression possibly with filtering (predicative) steps
{18} <computed_value>  ::= arithmetic combination of <variable>s and <constant_value>s
{19} <start_tag>       ::= '<'  <name>  '>'
{20} <end_tag>         ::= '</' <name>  '>'                       [<name> is a valid identifier for XML elements]
```

Figure 10. EBNF specification of the XQuery subrange which can be represented with XQBE

(and defined by) a path expression, which is derived from the path that in the graph reaches the node.

Variables are also instantiated in correspondence to *bifurcations* (nodes with multiple outgoing arcs). These variables will help in enforcing that the items in the branching paths do belong to *the same* common ancestor (as in q5).

Join nodes (those with *confluences*) originate as many variables as their incoming arcs (more than two are allowed, indeed). The predicative terms are generated as well, taking into account the comparator associated to the node.

Leaf nodes *with filtering labels* cause the instantiation of variables to express the selection conditions they are labelled with (within further predicative terms).

Negated branches are visited in the same way, with the only restriction that the visit does not begin until all the positive nodes have been visited, so as to guarantee that the scope constraints are not violated for the variables in the two-level clauses that will be generated ({5-13} in Figure 10).

3.2.2 Processing

A recursive visit of the *construct part* generates a FLWR expression for each node connected by a binding edge. The for clause of this expression defines the variable instantiated in 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 FLWR expressions if the short edge is below. In this second case, the for clause contains as many variables as the bound nodes reached by the outgoing arcs of the trapezoid.

This recursive visit of the construct part results thus in an XQuery statement composed of nested FLWR expressions. The where clause of each nested "internal" FLWR expression is assembled with the predicative terms that have been pre-computed in the preprocessing phase. These terms are collected out of the source part, according to topological criteria dictated by the variables already bound in the for clauses of the "external" FLWR expressions.

3.3 An example of translation

We now exemplify the translation process, applying it to query q8 and showing how the XQuery statement is generated, in the exact form in which it was shown in Section 2.7.

The preprocessing individuates four variables:

```
$b in doc("www.bn.com")//book  $a in $b/author
$f in $a/first/text()          $p in $b/publisher/text()
```

corresponding to the nodes labelled 'book' and 'author', and the PCDATA content of the 'publisher' and 'first' nodes. 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.

The processing starts its recursive visit of the construct part from the trapezium, and generates a couple of `<list>` tags to contain the reminder of the query (according to production {1} in the grammar of Figure 10). The recursive visit then moves to the `Jack-s-book` node, which is the vertex of a binding edge and thus originates a FLWR expression. The clauses of such expressions are built as follows.

The *for* clause binds only variable `$b`, associated to the other vertex of the binding edge (productions {3} and {4}).

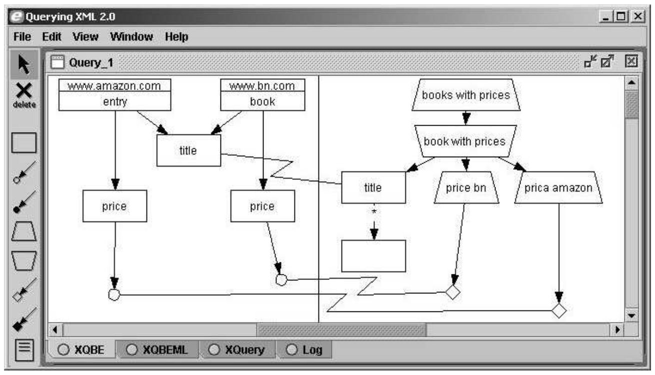

Figure 11. A snapshot of our interface

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 {5-13}:

```
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 according to production {14}, where production {15} intervenes to generate the <`Jack-s-book`> tags with its first disjunct and then a nested FLWR expression with its second disjunct. The decision to generate a nested FLWR expression depends on the fact that the <`FullName`> is bound by an edge. The algorithm recursively generates the nested FLWR expression binding `$a` and extracting only a subset of the conditions, and precisely that about `$f` (the choice is topologically defined, and depends on the fact that `$b` was already bound). Besides the nested FLWR expression the algorithm inserts a path expression that concludes the "breadth" projection ({15}, third disjunct) with the title, thus completing the translation.

Note that all the XQuery versions of queries q1 to q7 were presented as they are in the W3C Use Cases, for the sake of readability. Our algorithm translates the XQBE corresponding queries in a form that is equivalent but compatible with our grammar.

4 Our implementation of XQBE

This section briefly describes the features of our implementation of XQBE (a snapshot taken from the tool at work is shown in Figure 11).

Users draw queries in windows composed of two parts, corresponding to the source and construct parts. Graphs are built 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 and executed with a single click.

The graphical constructs and the graphs themselves are internally represented as XML data. XQBE queries can also be saved and exported as XML data.

Once the users complete their queries they can compile them to the corresponding XQuery statement, which is shown and can be executed on any XQuery engine. Intermediate results of the translation process are accessible as well. The translation is based on giving to all XQBE configurations a canonical (XML-based) representation, according to the algorithm sketched in Section 3.2.

The tool assists the user in many ways during the editing process, and provides syntactic feedback in several forms, to facilitate the drawing of correct queries. Many incorrect configurations are prevented "on line" by not allowing to connect two nodes or to draw a component in a place where it makes no sense. The syntactic feedback is not limited to "topological" errors, but makes default automatic and semi-automatic corrections to typical or frequent errors, both during the editing process and at compile time.

5 Related Work

Since the early times of XML, several textual query languages were proposed and analyzed by the database community [11, 13], far before the proposal of XQuery [25]. XQBE comes after a long stream of research on graph-based logical languages. The ideas in this field started years ago with the QBE paradigm [27].

The first graphical query languages were G [9] and G+ [10]. Graphlog [8] is a direct descendant of G+. A uniform notation for object databases where nodes represent objects and edges represent relationships was used in Good [21]. A Good-like notation was used by G-Log [14], a logic-based graphical language that allows to represent and query complex objects by means of directed labelled graphs. An evolution of this language, WG-Log [7], 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 [6], 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, consider again query q4, described in Section 2.2: this query, very easily expressible both in XQuery and XQBE, is not expressible in XML-GL (because of the ordering requirement). Moreover, XML-GL does not provide the capability to specify condition in the construct part, nor that of projecting “in depth” the extracted fragments.

QSByE (Query Semi-structured data By Example [12]) 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 [15]. Equix [4] 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 [5] a new syntax for Equix is proposed, more user-friendly but limited to searching the Web. BBQ [20, 16] (Blended Browsing and Querying) is a graphical user interface proposed for XMAS [18], 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 [3] (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 quey operations (such as simple selections, value based joins, universal quantification, negation, and complex predicates). VQBD [22] address the objective to explore an XML document of unknown structure.

XQForms [19] 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 [26] 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 that describe 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.

6 Conclusions and Future Work

In this paper, we presented 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 we deem fundamental to the success of XQuery for a wider audience. We have also presented a prototype that implements our proposed interface and translates graphical queries into XQuery, so that it can be used in conjunction with any XQuery engine. Among the implementations of XQuery listed by the W3C, we are currently using IPSI-XQ [1].

There are several potential opportunities for future work. From a technical viewpoint, several improvements and extensions to our work are possible, and among the various uncovered features of XQuery we will give priority to adding support for the XML typing system, extending the set of supported core functions and adding the capability of querying the document order.

Another opportunity is that of specializing XQBE and the tool 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 currently 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. The opportunity to alternate among the QBE and SQL representations makes QBE a valuable tool in MS Access.

Acknowledgements

In this work the authors are supported by the “Virtual Campus” Microsoft Research Grant.

References

[1] Ipsi-xq - the xquery demonstrator, 2003. `http://ipsi.fhg.de/oasys/projects/ipsi-xq/index_e.html`.

[2] L. Bouganim, T. Chan-Sine-Ying, Tuyet-Tram Dang-Ngoc, J. L. Darroux, G. Gardarin, and F. Sha.

Miro web: Integrating multiple data sources through semistructured data types. In *Proc. of 25th Int. Conf. on Very Large Data Bases (VLDB'99), Edinburgh, Scotland, UK*, pages 750–753, Sept. 1999.

[3] M. Carey, L. Haas, V. Maganty, and J. Williams. Pesto: An integrated query/browser for object databases. In D. McLeod, R. Sacks-Davis, and H. Schek, editors, *Proc. of the 22nd Int. Conf. on Very Large Databases (VLDB)*, pages 203–214, 1996.

[4] S. Cohen, Y. Kanza, Y. A. Kogan, W. Nutt, Y. Sagiv, and A. Serebrenik. Equix easy querying in XML databases. In *WebDB (Informal Proceedings)*, pages 43–48, 1999.

[5] 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*, Sept 2000.

[6] S. Comai, E. Damiani, and P. Fraternali. Computing graphical queries over xml data. *ACM TOIS*, 19(4):371–430, 2001.

[7] S. Comai, E. Damiani, R. Posenato, and L. Tanca. A schema based approach to modeling and querying www data. In *FQAS'98*, May 1998.

[8] Mariano P. Consens and Alberto O. Mendelzon. The graphlog visual query system, 1990.

[9] Isabel F. Cruz, Alberto O. Mendelzon, and Peter T. Wood. A graphical query language supporting recursion, 1987.

[10] Isabel F. Cruz, Alberto O. Mendelzon, and Peter T. Wood. G+: Recursive queries without recursion. In *2nd Int. Conf. on Expert Database Systems*, pages 355–368, 1988.

[11] 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. Xml query languages: Experiences and exemplars. 1999.

[12] Irna M. R. Evangelista Filha, Alberto H. F. Laender, and Altigran S. da Silva. Querying semistructured data by example: The qsbye interface.

[13] Z. G. Ives and Y. Lu. Xml query languages in practice: an evaluation. In *WAIM'00*, 2000.

[14] P. Peelman J. Paredaens and L. Tanca. G-log a declarative graph-based language. *IEEE Trans. on Knowledge and Data Eng.*, 1995.

[15] 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*, pages 124–138, 1982.

[16] Y. Papakonstantinou K. Munroe. Bbq: A visual interface for browsing and querying xml, 2000.

[17] S. Kepser. A proof of the turing-completeness of xslt and xquery. Technical report SFB 441, Eberhard Karls Universitat Tubingen, May 2002.

[18] B. Ludaescher, Y. Papakonstantinou, P. Velikhov, and V. Vianu. View definition and dtd inference for xml. In *Proc. Post-IDCT Workshop*, 1999.

[19] Y. Papakonstantinou M. Petropoulos, V. Vassalos. Xml query forms (xqforms): Declarative specification of xml query interfaces. In *Proc. of WWW10*, 2001.

[20] K. Munroe, B. Ludäscher, and Y. Papakonstantinou. Blended browsing and querying of xml in a lazy mediator system, March 2000.

[21] J. Paredaens, J. Van den Bussche, M. Andries, M. Gemis, M. Gyssens, I. Thyssens, D. Van Gucht, V. Sarathy, and L. V. Saxton. An overview of good. *SIGMOD Record*, 21(1):25–31, 1992.

[22] J Yeo S. Chawathe, T. Baby. Vqbd: Exploring semistructured data (demonstration description). In *Proc. of the ACM SIGMOD Int. Conf. on Management of Data*, 2001.

[23] W3C. Extensible Stylesheet Language (XSL). `http://www.w3c.org/TR/xsl/`, October 2001.

[24] W3C. XML Query Use Cases. http://www.w3.org/TR/xmlquery-use-cases, November 2002.

[25] W3C. XQuery: An XML Query Language. http://www.w3.org/XML/Query, November 2002.

[26] V. Vassalos Y. Papakonstantinou, M. Petropoulos. Qursed: Querying and reporting semistructured data. In *Proc. of the ACM SIGMOD Int. Conf. on Management of Data*, 2002.

[27] Moshé M. Zloof. Query-by-example: A data base language. *IBM Systems Journal*, 16(4):324–343, 1977.