An Abstract of the Project of

Shiwei Zhao for the degree of Master Science in Computer Science presented on July 15, 2001.

Title: A Visual XML Query Interface

Abstract approved: _____

Martin Erwig

As XML becomes more and more popular, easy-to-use and powerful XML query languages are in great need. Xing is a visual query and restructuring language for XML documents. The objective of this project is to develop a basic version of Xing, including a user-oriented XML query interface and a simple query translation implementation. The interface design is based on a visual document metaphor and the notion of document patterns and rules. End users do not have to be good at programming or XML syntax to use Xing. In this report, we describe the implementation for the Xing prototype, including GUI design, data structures and algorithms. We also compare the features of Xing with other XML query languages.

A Visual XML Query Interface

by

Shiwei Zhao

A Project

submitted to

Oregon State University

In partial fulfillment of
the requirements for the
degree of

Master of Science

Completed July 15, 2001

Acknowledgments

I am grateful to my major professor, Dr. Erwig, for the principal ideas, helpful hints, precious comments he puts in my work. This project was extensively discussed in a series of meetings with him. My appreciation also goes to Dr. Burnett and Dr. Bose for their help and being on my committee. Thanks also go to all my friends who gave me their valuable suggestions and encouragements.

Table of Contents

List of Figures

Page

A Visual XML Query Interface

Chapter 1: Introduction

In recent years, XML [10] is becoming more and more popular. People in diverse fields started to realize its great simplicity, generality, and standardization on data description, data exchange and web representation. To utilize its great features on data processing, the need for XML query languages emerges, and lots of different XML query languages have been developed. However, one realistic problem is that most existing query languages are too complex, maybe not for professionals, but for some end users who need to query XML databases while not being professionals.

Thus, there has been research for expressing XML queries in a user-friendlier notation: "a visual query and restructuring language for XML documents, called Xing (which is pronounced 'crossing' and which is an acronym for **X**ML **in** **G**raphics)" [1]. In this project, we developed a user-oriented XML query interface, which is based on "a visual document metaphor and the notion of document patterns and rules" [1], and we implemented a simple query translation.

1.1 XML

XML is a language for creating markup languages, which describe data in a format with no indication of how the data is to be displayed. XML is rapidly developing into the standard format for data exchange and for representing data on the Internet. More and more companies and researchers start to apply XML for their product or research.

XML data is fundamentally different from relational or object-oriented data. Therefore neither SQL nor OQL is appropriate for XML. Because XML is self-describing and not rigidly structured, it has some important abilities that do not exist in relational or object-oriented data, like the ability to model irregularities naturally. XML can be used as database or

work with other databases. For enterprises trying to meld incompatible systems, XML can serve as a common transport technology for moving data around in a system-neutral format. In addition, XML can handle all kinds of data, including text, images, and sound, and is user extensible to handle anything special. Based on all those merits, we believe that XML can be applied much better than other data models on data extraction, conversion, transformation, and integration, whose solutions rely on a *query language.*

An XML document primarily consists of a strictly nested hierarchy of elements with a single root. A Document Type Definition (DTD) [10] can be used to define the document structure so that the documents can be checked to conform to a predefined format. A DTD can be declared inline in a XML document, or as an external reference.

1.2 Visual Programming Language (VPL)

"When a programming language's (semantically-significant) syntax includes visual expressions, the programming language is a *visual programming language* (VPL)" [3]. The goals of VPLs include making programming easier for humans and helping achieve better programming language design.

1.3 Xing and Design Goals

There have been lots of different query language drafts developed, which emphasize different features. However, a big problem for beginning end users who want to do queries on XML databases may be that they do not know those sophisticated XML query language, and it will be so painful to study since they may not be professionals in computer technology. So in that case it is in need that there is a user-friendly query interface to provide an easy and direct way to express those queries. We hope that Xing [1, 4] can help to get the power.

In Xing, users only need to draw forms to mimic the structured XML document. Users can start with very simple searches and can advance by adding structural requirements. In addition to simple queries based on document patterns, reformatting and restructuring of query results is also possible.

1.4 Organization

In this section, we have introduced XML, VPL, and the design goal of our Xing system. In Chapter 2, related work in XML query languages and in visual XML query languages is reviewed. In Chapter 3, it is discussed how to use Xing and how to pose queries in Xing. Translation from Xing to XML Query Algebra is the content of Chapter 4. Design issues of the Xing prototype, including GUI design, data structures, several important algorithms exploited in the implementation, are described in Chapter 5. Conclusions and possible future work follow in Chapter 6.

Chapter 2: Related Work

2.1 XML Query Languages

As an increasing amount of information is stored, exchanged, and presented using XML, the need for retrieving the information efficiently emerges. Since XML has great flexibility in representing different kinds of data, including the information traditionally considered to be a database or to be a document, a very important issue in the XML application world is how to intelligently query the diverse data sources.

To satisfy the increasing need for better query ability, there have been lots of different XML query languages proposed, most of which are based on textual notations. A few of them are introduced below.

2.1.1 XML Query Algebra

XML Query Algebra [2] is proposed by the W3C XML Query Working Group [15]. The group has defined query requirements [11] and a data model [12] for XML documents. The data model is based on the W3C XML information set [17] that provides a consistent set of definitions of the information in a well-formed XML document. It is also the foundation of the W3C XML Query Algebra. The group hopes to use the algebra both to supply well-defined query semantics and to support query optimization. So it has enough power and compact expression.

Because XML Query Algebra is well defined and well designed, it is well suited to verify and execute Xing queries. Therefore, we think that the algebra is a good textual target representation for our visual queries. We will consider XML Query Algebra in detail in Chapter 4.

2.1.2 XQuery

XQuery [6] is also designed by the W3C XML Query Working Group. The group hopes to make it meet the requirement for a human-readable query syntax, that is to keep it as a small, easy-to-implement language in which queries are concise and easily understood. It is derived from an XML query language called Quilt [18]. XQuery is a functional language, which may take several different forms of expression to implement the same query. XQuery is otherwise very similar to XML Query Algebra.

2.1.3 XML-QL

The main concern of XML-QL [5] is about large data repositories, heterogeneous data integration, legacy data export, and data transformation. An XML-QL query usually includes three clauses: (1) pattern: to match nested elements in the input document and bind variables; (2) filter: to test the bound variables; (3) constructor: to specify the result in terms of the bound variables.

To give an impression of the style of expressing queries in XML-QL, we show here two examples. The same queries will later be examined in Xing and XML Query Algebra. The first example selects all *book* elements from a bibliography database: *www.xml.com/bib.xml*.

```
CONSTRUCT <bib> {
   WHERE
            <bib>
                  <book> $b </book>
            </bib> IN "www.xml.com/bib.xml"
   CONSTRUCT
            <book> $b </book>
} </bib>
```

In XML-QL queries, patterns and filters appear in the WHERE clause, and the result expressions appear in the CONSTRUCT clause. The WHERE clause generates a relation that maps variables to tuples of values that satisfy the clause. Then, the CONSTRUCT clause constructs elements for every tuple that satisfies the WHERE clause.

The second example illustrates how to specify some constraints on the elements that appear in the result.

```
CONSTRUCT <bib> {
  WHERE
        <bib>
              <book year=$y>
                    <title>$t</title>
                    <publisher>Addison-Wesley</publisher>
              </book>
        </bib> IN "www.xml.com/bib.xml",
        $y > 1991
  CONSTRUCT <book year=$y><title>$t</title></book>
} </bib>
```

XML-QL's structure is similar to an XML document, and thus easy to understand for people that know XML syntax.

2.2 Visual XML Query Languages

Besides Xing, there are also two other visual XML query languages in development, with different emphasis.

XML-GL [8] is a language proposal that uses a graph-based formalism. The tree and graph representation used by XML-GL is well suited for a general-purpose formal language manipulation. However, it is questionable whether XML-GL is simple enough for a user-oriented query language. In our opinion, to represent XML documents in a form-like way can still reflect their tree structure, whereas at the same time it is better suited for the user's view and more convenient for object representation and the user's operation.

Figure 2.1 is a simple query in XML-GL, which selects all the *book* elements of a particular author from a document and list the titles and the authors. The left-hand-side graph contains the extract part of the query. The right-hand-side graph contains the clip part, which defines the DTD of the result document.

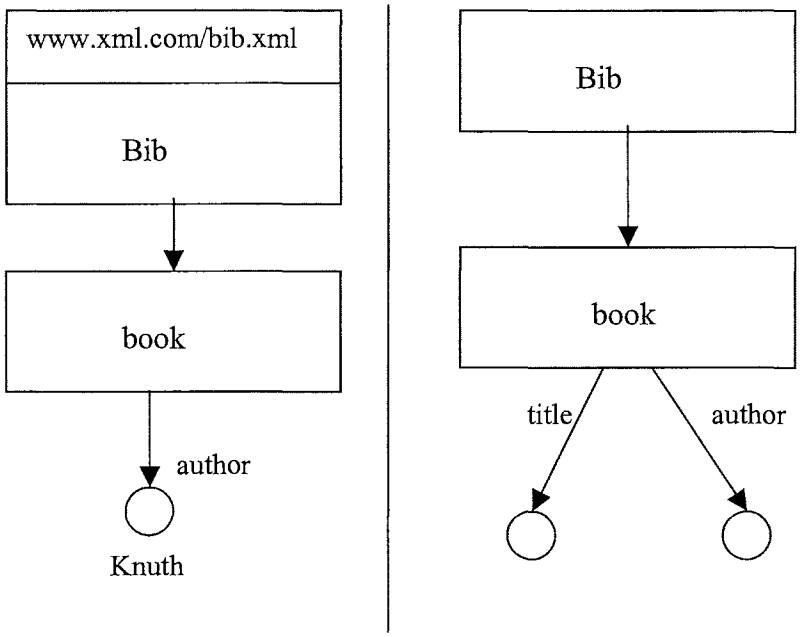

Figure 2.1. A sample XML-GL query

There is also another form-based query interface provided by EquiX [9]. Its form-based GUI is constructed automatically from the DTD of XML documents, which does not require any intervention from users. EquiX also generates automatically a DTD for the result documents, which is helpful for users, too. However, it is also very restricted since only XML data having a DTD can be operated. Another limitation is that it does not allow users to reformat the query results, which limits its application area.

To give an impression of the style of EquiX queries, we copy an example from [9] because EquiX is currently not accessible for us to create a query. The example is presented in Figure 2.2, which computes the courses not taught by Dr. Jekyll.

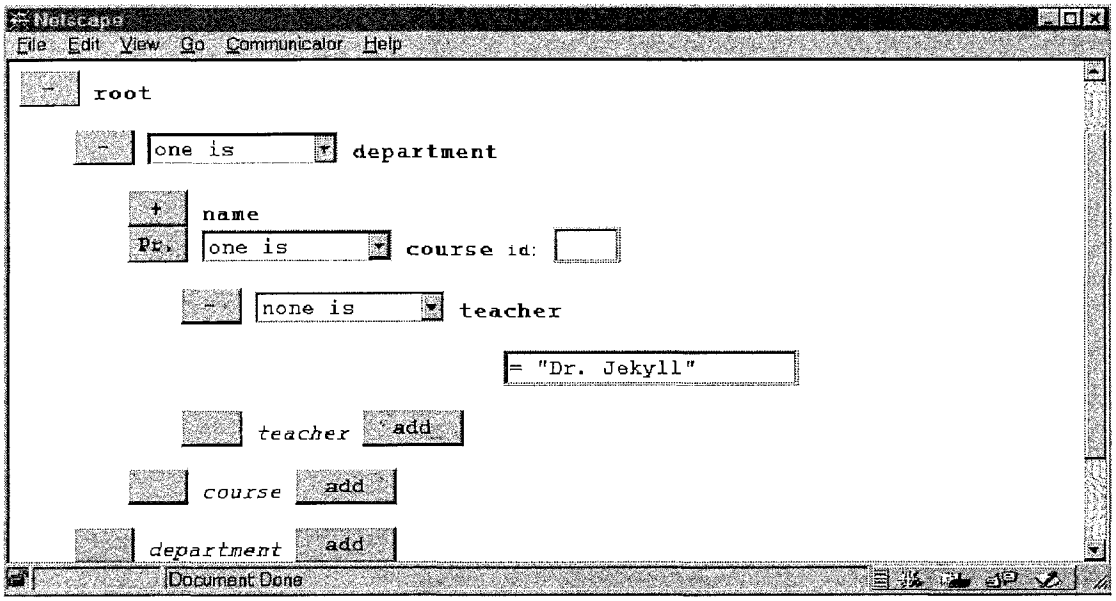

Figure 2.2. A sample EquiX query

Chapter 3: The Xing System

3.1 How It Works

The program can be run from a web browser or as a standalone application. It is a menu-based, event-driven system. The main purposes are a user-friendly interface and an easy-to-understand visual object representation.

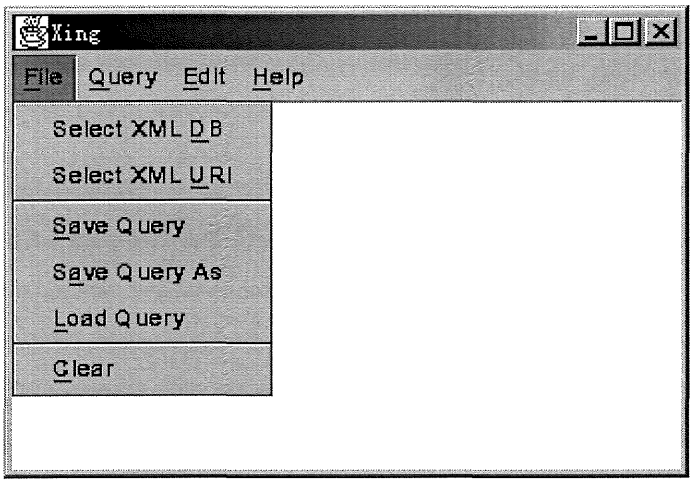

Figure 3.1. File menu in Xing

First, to create a query, users must specify an XML database to work on. If run as an application, you can select a local XML document or specify a URI for an XML document. If run as an applet from a browser, then, for security reasons, local files cannot be operated on. From the File menu, see Figure 3.1, a user can also save a query or load an existing query when run as an application. The *Clear* menu item is used to clear the screen and restore the internal status before making a new query.

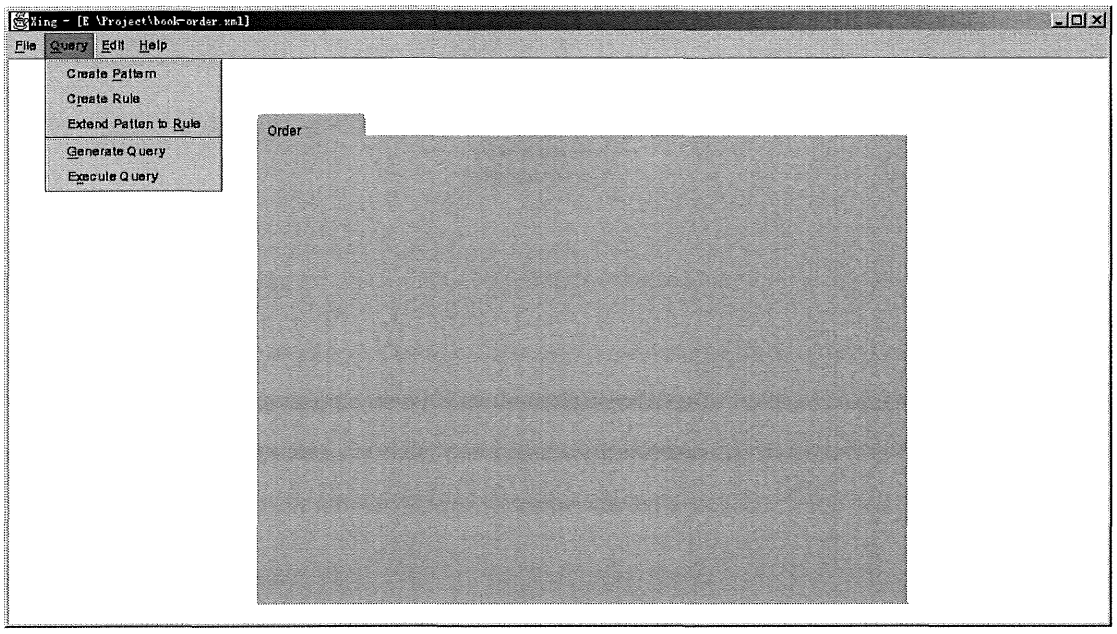

Figure 3.2. Query menu in Xing

In the second step, users may start to draw a visual query. After the XML document is selected, a folder icon with a specified root name shows up automatically for users' convenience, see Figure 3.2. Users can also take their own option from the *Query* menu: select *Create Pattern* to make simple queries, or select *Create Rule* to express more complex features. A document rule consists of two document patterns that are joined by a double arrow. *Extend Pattern to Rule* may be used when users are partly or fully done with editing their pattern and then find they need a rule to express a more complicated data transformation. In this way, users do not lose their previous work when creating rules. From this menu, *Generate Query* is used to translate a visual query into a textual query, an expression of XML Query Algebra, and *Execute Query* is used to run the query on an XML data source. The current version does not include the implementation of a query execution engine and can only translate pattern-represented queries.

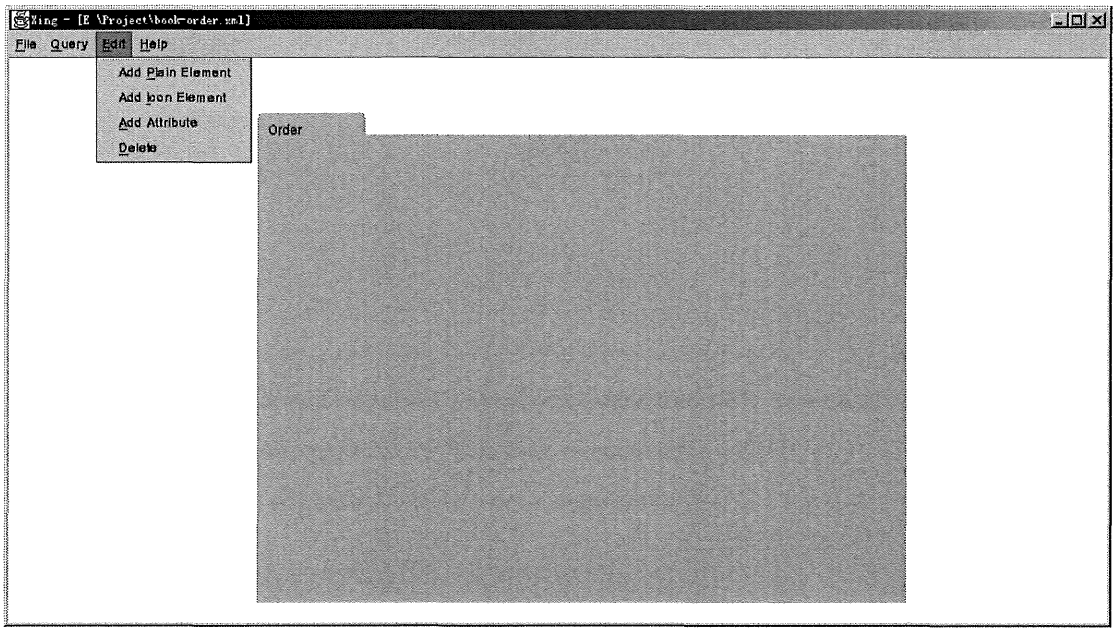

Figure 3.3. Edit menu in Xing

After users have created a folder to represent the root of the XML document, they can use the *Edit* submenu to add more components on the folder to constrain the query, see Figure 3.3. An icon element represents a structured element (that has subelements and attributes as children), see Figures 3.4 & 3.5. Users can supply constraints on plain elements (that has no any children) and attributes to specify the query. Plain elements and attributes can only be added to parent icon elements. To delete a component from the current query, users may click on the component and select *Delete* menu item or click <Delete> key.

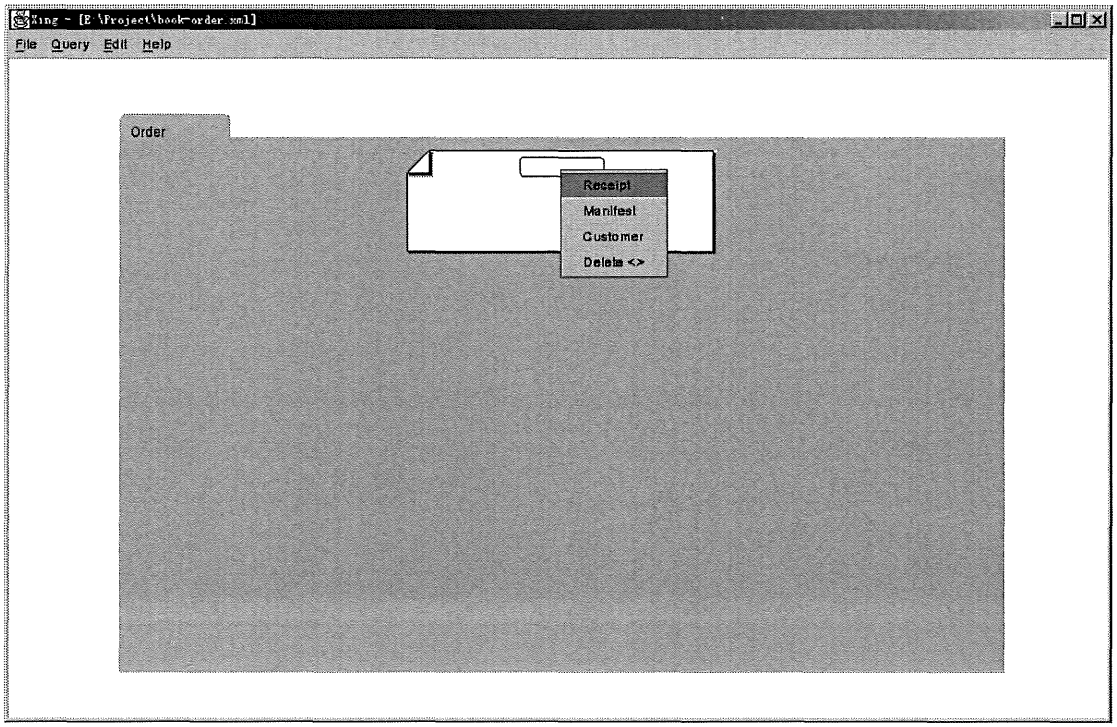

Figure 3.4. Right-click pop-up menu on a text field

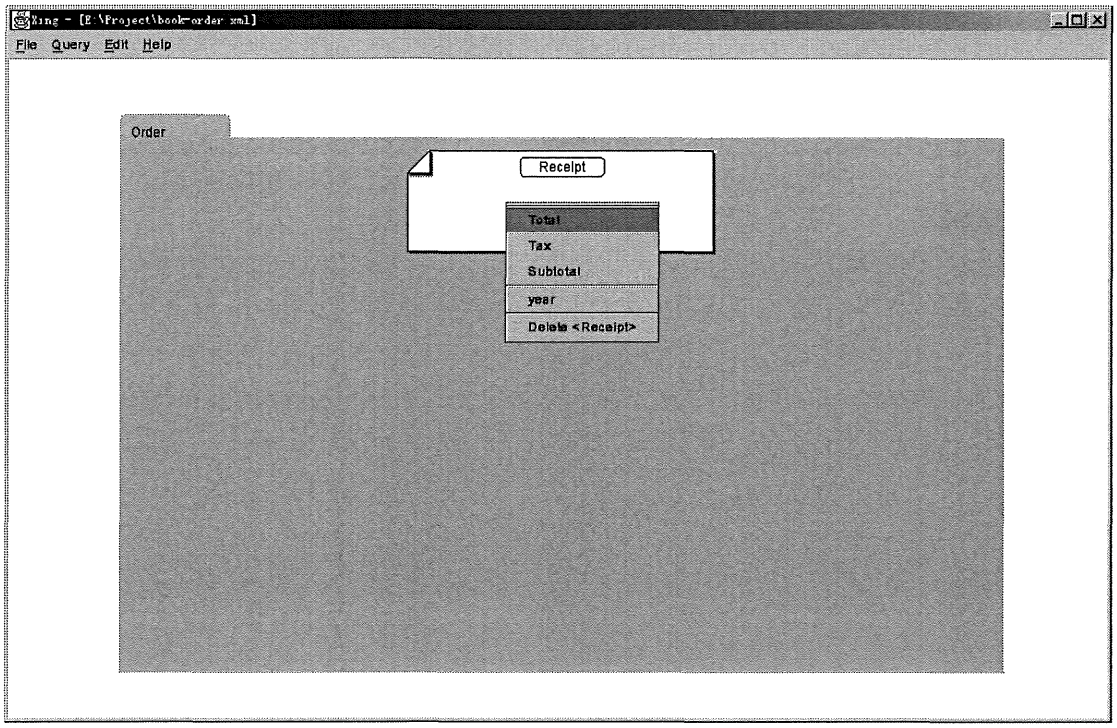

Figure 3.5. Right-click pop-up menu on an icon or folder

If the XML document contains (a link to) a DTD to define the document structure, our system supplies a very convenient and direct option to edit the query. If users click the right mouse button on any text field (Figure 3.4) or icon (Figure 3.5), all possible names that can be added will be listed in a pop-up menu. When an icon is clicked on, the menu items may include the names of structured elements, plain elements, and attributes. The items are separated by category. As Figure 3.5 shows, the *Receipt* icon can include no structured element, three plain elements, and one attribute. Users may select the Delete <...> item to delete the focused component, and may click any other item to insert the item name to the focused text field or to insert a component with the item name to the focused icon or folder.

3.2 How to Create Queries

To illustrate what kinds of queries we can create with Xing and how to express them with Xing, we describe some typical XML query scenarios here, including selection,

flattening, restructuring, regrouping, and so on. In the next chapter, we will represent every scenario with a textual query language, XML Query Algebra, to compare the two languages and, in particular, to show Xing's advantages and limits.

We use the following running example. Assume the XML input is in the document www.xml.com/bib.xml, containing bibliography entries described by the following DTD.

```
<!ELEMENT bib  (book*, article*)>
<!ELEMENT book  (title,  (author+ | editor+ ), publisher, price )>
<!ATTLIST book  year CDATA  #REQUIRED >
<!ELEMENT article  (title,  (author+ | editor+ )) >
<!ATTLIST article  year CDATA  #REQUIRED >
<!ELEMENT author  (#PCDATA )>
<!ELEMENT editor  (#PCDATA )>
<!ELEMENT title  (#PCDATA )>
<!ELEMENT publisher  (#PCDATA )>
<!ELEMENT price  (#PCDATA )>
```

This DTD specifies two different elements: *book* or *article*. A *book* element contains one *title* element, one or more *author* elements or one or more *editor* elements, one *publisher* element, and one *price* element; it also has a *year* attribute. An *article* element contains only one *title* element, and one or more *author* elements or one or more *editor* elements. A *title*, *author*, *editor*, *publisher*, or *price* element contains text.

In the following sections, all of our example queries will be based on this document.

3.2.1 Using Patterns

The first example is the simplest. We just select all books, by checking the element tag (see Query 1 in Figure 3.6). Since we keep the result's original structure, a pattern is sufficient for our needs.

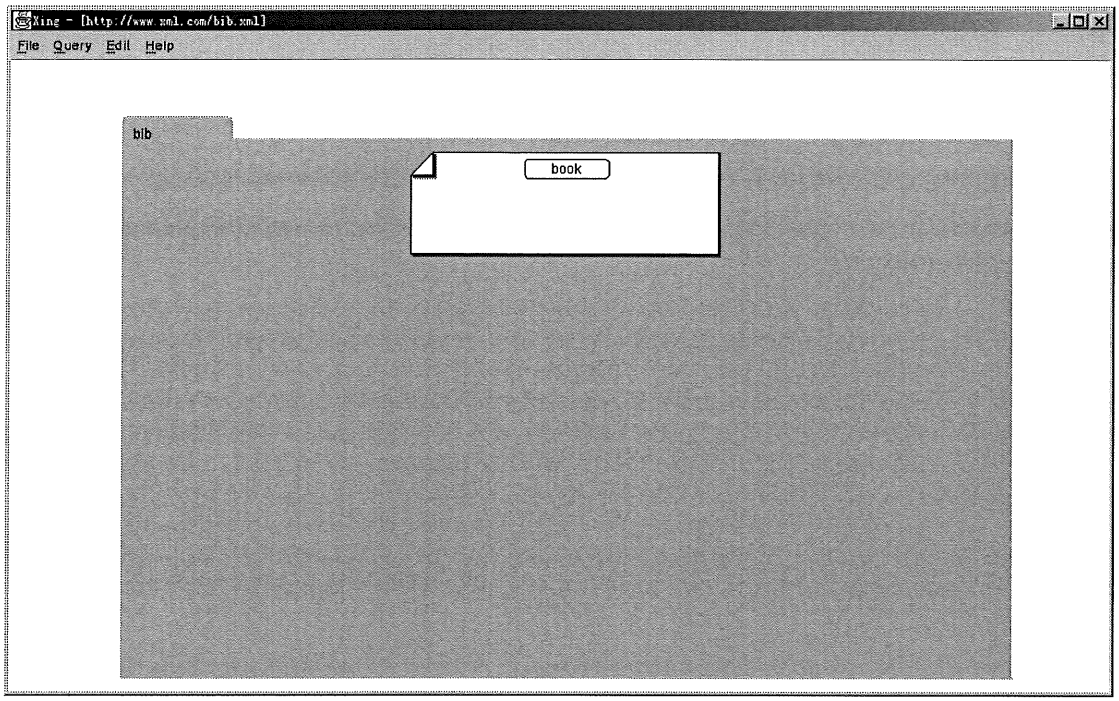

Figure 3.6. Query 1: selecting all subelement with tag name *book*

Here we use a folder icon to represent the root element, inside of which multiple-layered document icons are used to represent subelement.

Next, let us consider a little more complex query: select all publications, either books or articles, of a particular author, and list the titles and the author (see Query 2 in Figure 3.7). Here, we have to check some properties of the subelements.

For example, assume there are two elements in our sample document, which are listed as follows:

```
<book year = "1992">
    <title> Concrete Mathematics </title>
    <author> Graham </author>
    <author> Knuth </author>
    <publisher> Addison-Wesley </publisher>
    <price> 65.95 </price>
</book>
<article year = "1998">
    <title> Introduction to Linear Algebra </title>
    <author> Knuth </author>
</article>
```

We want to list all titles written by author 'Knuth', and expect to get the elements:

```
<book>
    <title> Concrete Mathematics </title>
    <author> Knuth </author>
</book>
<article>
    <title> Introduction to Linear Algebra </title>
    <author> Knuth </author>
</article>
```

Here, other authors of the same *book* or *article*, if any, are excluded.

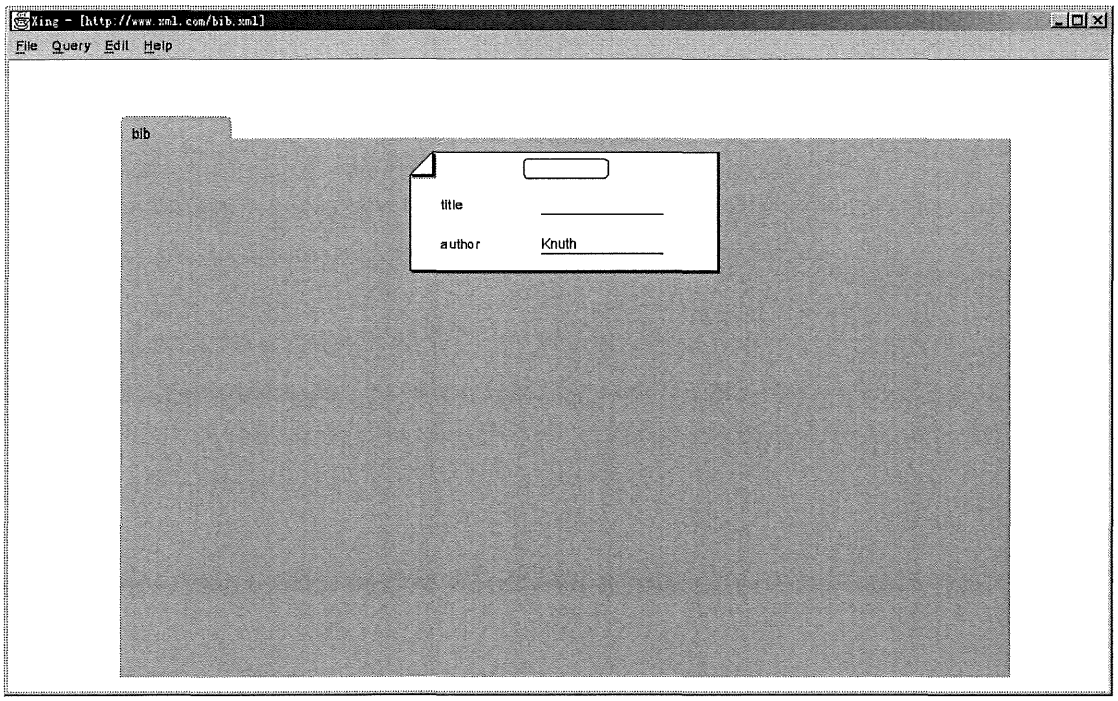

Figure 3.7. Query 2: selecting all elements whose subelements *author* include "Knuth"

Inside of the document icon, users only need to insert two plain elements and specify their tag names, which may or may not have constraints, to define what information to extract. The empty title tag of the document icon represents an expression like the wildcard, which means we want to find all structured elements, whose children include *title* and *author* elements. In the condition field of *author* element, a specific name 'Knuth' is typed to express the constraint that only those books or articles written by 'Knuth' appear in the result.

3.2.2 Using Rules

In the first two examples, document patterns were sufficient to express the queries. Document patterns perform selection first, followed by a projection consistent with the pattern. However, we often need to do a projection that is not consistent with the selection pattern, so we have to employ a new pattern to form a rule. In later examples, we also have to employ document rules to achieve restructuring.

Our next example is Query 3 in Figure 3.8, which selects all books published by Addison-Wesley after 1991. Moreover, the *book* elements in the result only include their *year* and *title*.

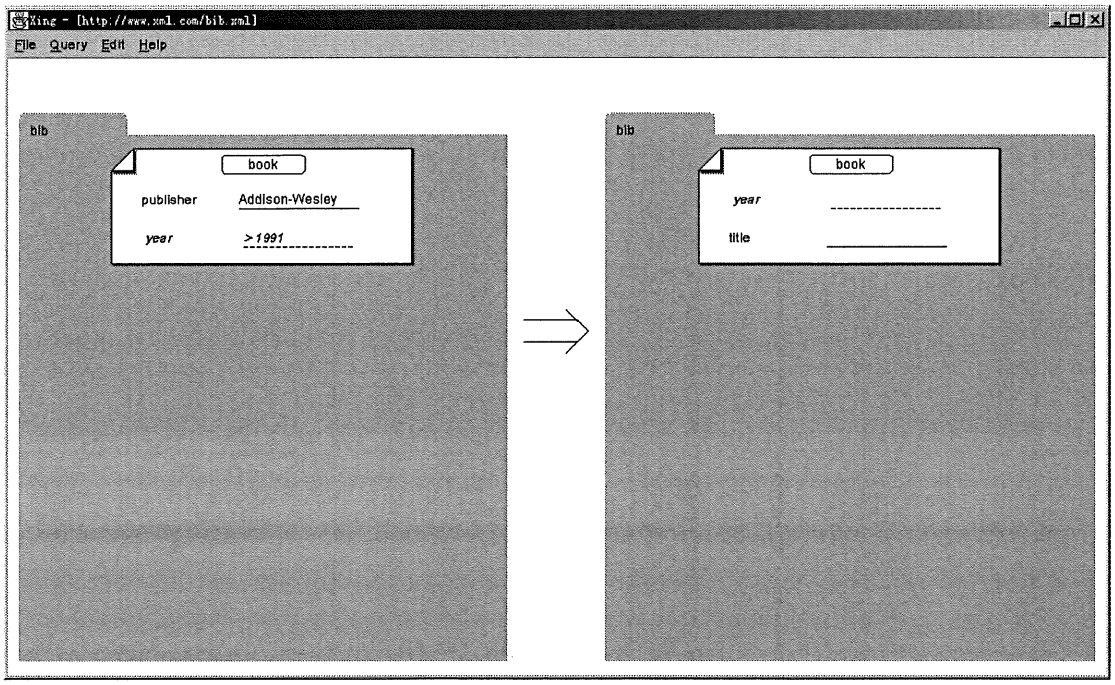

Figure 3.8. Query 3: selection and extraction

In Xing we display attributes and basic elements with different fonts and base lines, as shown in Figure 3.8.

The next query preserves the grouping of results by title (see Query 4 in Figure 3.9 & 3.10). At the same time, we also change the tag name from 'book' to 'result'. Thus, using the same sample data source above, we expect to get the result shown below:

```
<result>
        <title> Concrete Mathematics </title>
        <author> Graham </author>
        <author> Knuth </author>
</result>
```

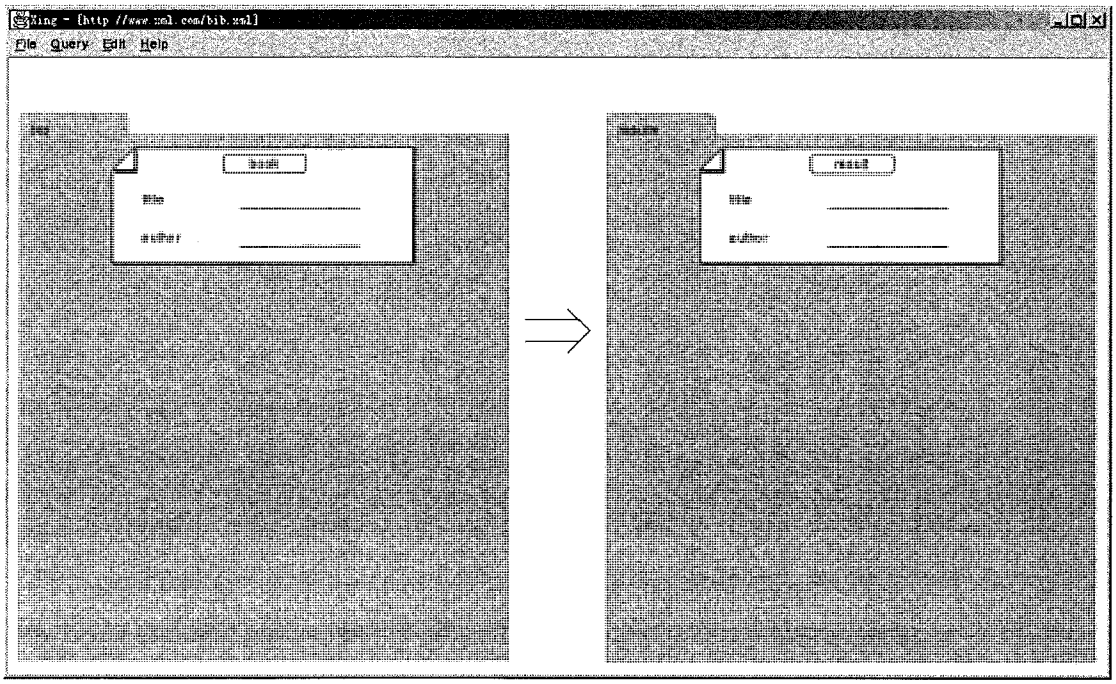

Figure 3.9. Query 4: preserving structures and change tag name: the children of *book* icon have no constraints

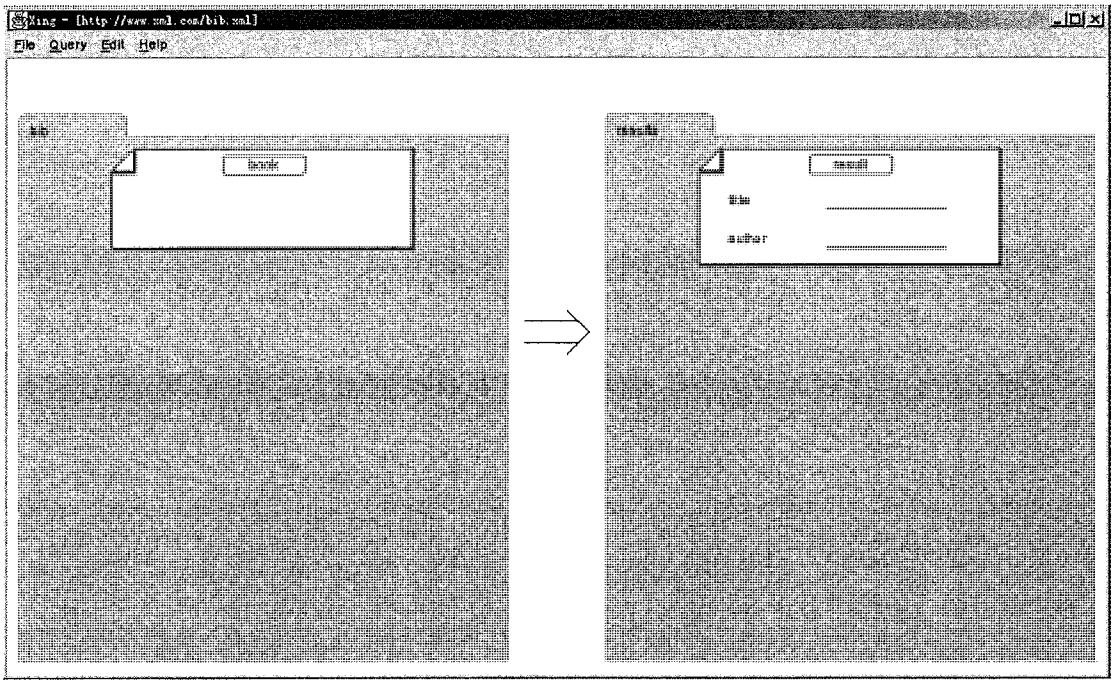

Figure 3.10. Query 4: preserving structures and change tag name: *book* icon is empty

According to Xing's semantics, if there are no constraints for the elements in the pattern, those elements can as well be omitted from the pattern. Thus, the queries in Figure 3.9 & 3.10 are identical with regard to their semantics. For every book all authors are matched and returned in one tuple.

Query 5 in Figure 3.11 regroups the database by authors to get a list of authors, and lists every book of theirs with title and publisher.

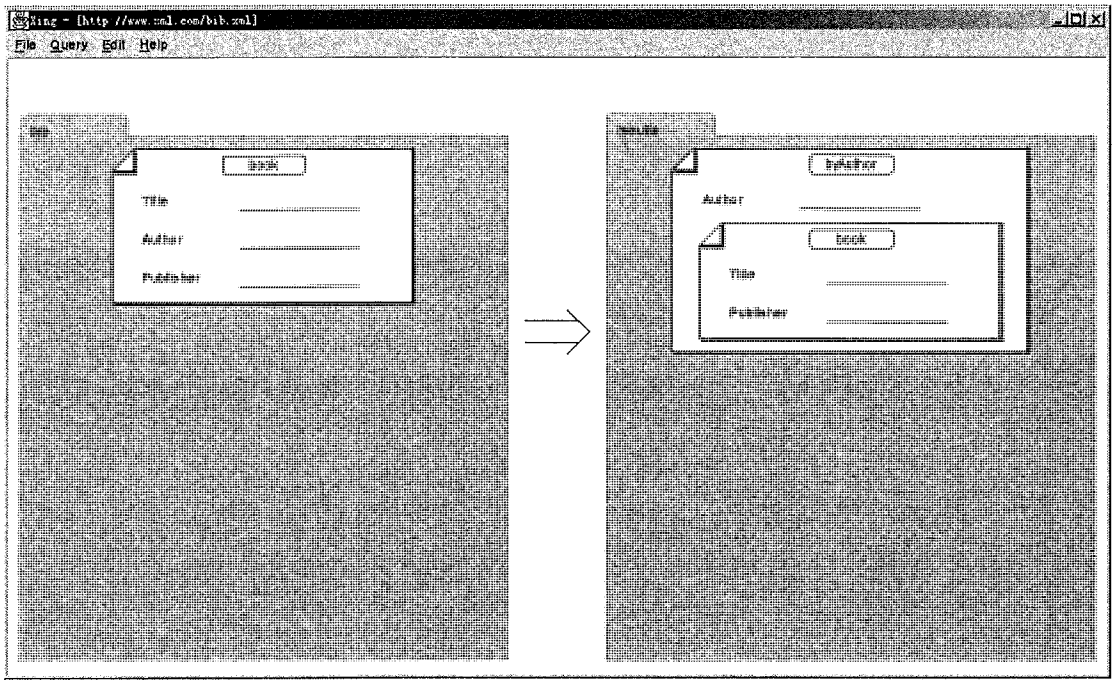

Figure 3.11. Query 5: restructuring transformation

3.2.3 Queries that Xing Cannot Express

We note that the current version of Xing is still limited in its expressiveness. In this section, we list several general query examples that cannot currently be expressed in Xing. Then in the next chapter, the solution in XML Query Algebra will be discussed.

The previous example, Query 4, preserves the structure of the data source. The next query, we call it Query 6, returns a collection of all title-author pairs of *book* elements. The query flattens the nested structure, each book contributing one pair for each author.

Using the same sample data source, we hope to get two different elements related to the two authors separately:

```
<book>
        <title> Concrete Mathematics </title>
        <author> Graham </author>
</book>
<book>
        <title> Concrete Mathematics </title>
        <author> Knuth </author>
</book>
```

This query cannot be expressed in Xing. It actually requires us to restructure the data source, so we have to use rules here, instead of patterns, to explicitly express the result's structure. Unfortunately, the current version of Xing does not support this particular form of restructuring. One possible solution and a discussion can be found in Sections 3.3 & 4 of [2]. The design of Xing and the corresponding semantics is still under development.

Another typical query is to illustrate restructuring by grouping each author with the titles he or she has written (Query 7). This requires joining elements on their *author* values. Although Xing can perform grouping operations with a grouping element like in Query 5, it cannot express this query because it cannot construct a variable number of *title* elements on the same level as one *author* element.

Sometimes, it is required to extract information from multiple documents and merge them. In Xing, for simplicity, users can only select one data source to construct a query on it at one time, so it does not currently support a combination of multiple data sources. On the other hand, it is not impossible to implement it in a visual language. Interested readers can find a discussion of this issue in [1].

Elements in an XML document are ordered. In some cases, it might be important to refer to the order in which elements appear, to preserve the order in the output, or to impose a new order. However, referring to individual elements by position is not possible in Xing. To represent a sorting operation is also too complicated for Xing's current version.

Chapter 4: XML Query Algebra

In this chapter, we use XML Query Algebra to create the queries for the same scenarios explored in Section 3.2 and try to identify the relationship between Xing and XML Query Algebra. Meanwhile, in every query, different features of XML Query Algebra are explained.

To present the queries according to their syntax features, we explore the scenarios in a different order from Section 3.2.

4.1 Projection

One of the most basic operations of XML Query Algebra is projection. The algebra uses notations that have similar syntax and meaning to XPath's [19] path navigation.

The Query 1 from Figure 3.6 can be expressed in XML Query Algebra as follows:

```
Let bib0 = ("www.xml.com/bib.xml")/bib
           bib [bib0/book]
```

The *let* expression in the above query binds the variable *bib0* to the root of an XML document, so that the global variable *bib0* may be used in any other expression of the query. The document is given by URI enclosed in quotes, where the root element's name follows with a slash. Then the second expression represents the query to be executed. There, *bib* is used as the tag name of the newly constructed element, whose children are enclosed in brackets (the notation [...]). And *bib0/book* represents the projection operation to return all its subelements with the element name *book*.

4.2 Iteration

Another common operation is the *for* expression, which is used to iterate over all subelements of one element and to bind a variable to each such element. Then the content of elements can be accessed later and can be transformed into a new content. For example, the

XML Query Algebra expression for Query 4 in Section 3.2.2 includes iteration and projection operations:

```
Let bib0 = ("www.xml.com/bib.xml")/bib
results [
   for b in bib0/book do
            result [b/title, b/author]
]
```

Let us look at Query 6 in Section 3.2.3. It cannot be expressed in Xing, but it can be expressed in XML Query Algebra.

```
Let bib0 = ("www.xml.com/bib.xml")/bib
bib [
    for b in bib0/book do
    for a in b/author do
            book [b/title, a]
]
```

Actually, the typing rule of the *for* expressions in XML Query Algebra is rather involved. Comparing between the above two algebra expressions, you may find that there are only small differences in the algebra expressions although the query results will be quite different. Recall the query result discussion in Section 3.2. In Query 4, for every book element we get a result element that consists of the book's title and all its authors.

```
<result>
            <title> Concrete Mathematics </title>
            <author> Graham </author>
            <author> Knuth </author>
</result>
```

In contrast, in Query 6, for every author element, we get one *book* element for each author.

```
<book>
            <title> Concrete Mathematics </title>
            <author> Graham </author>
</book>
<book>
            <title> Concrete Mathematics </title>
            <author> Knuth </author>
</book>
```

4.3 Selection

To select values that satisfy some predicate, we use the *where* expression. Using XML Query Algebra to express Query 2 in Figure 3.7, we get:

```
Let bib0 = ("www.xml.com/bib.xml")/bib
bib [
for b in bib0/* do
        for a in b/author do
                where a/data() = "Knuth" do
                        b [b/title, a]
]
```

The wildcard (*) in the expression means that we want to go through all elements that are the children of the element that *bib0* points to. Thus, in this case, the outermost *for* clause iterates over all subelements of the root element and binds the variable b to each such element, and the inner *for* clause iterates over all subelements of b and only binds the variable a to each element with tag name *author*. The outermost *for* clause determines the order of the result, which means that all *book* elements will be listed before the *article* elements in the result according to their order in the XML data tree. The *where* clause is used to select any element whose value is "Knuth". The built-in function *data()* is used to access atomic data of one basic element like strings or integers.

As can be seen in Figure 3.8, Xing displays attributes and basic elements with different fonts and base lines. In XML Query Algebra an @ sign is added before an attribute name to tell it from an element name. Below is the XML Query Algebra expression for Query 3:

```
Let bib0 = ("www.xml.com/bib.xml")/bib
bib [
for b in bib0/book do
   where b/publisher/data() = "Addison-Wesley" do
      where b/@year/data() > 1991 do
            book [b/@year, b/title]
]
```

4.4 Special Functions

XML Query Algebra supplies a set of built-in functions. To express Query 5 in Figure 3.11, a built-in function *distinct_value()* has to be employed, which produces a forest of nodes whose values are all distinct. The complete code is listed below:

```
Let bib0 = ("www.xml.com/bib.xml")/bib
results [
for a in distinct_value (bib0/book/author/data()) do
        byAuthor [
            author [a],
            for b in bib0/book do
                        for a2 in b/author/data() do
                        where a = a2 do
                                    book [b/title, b/publisher]
        ]
]
```

Here, for every distinct *author* a new element *byAuthor* is created. The brackets can be nested to make a multi-layer structure transformation.

To express Query 7, which requires joining elements on their *author* values, the *distinct_value()* is also employed.

```
Let bib0 = ("www.xml.com/bib.xml")/bib
results [
for a in distinct_value(bib0/book/author/data()) do
    result [
            author[a],
            for b in bib0/book do
                for a2 in b/author/data() do
                    where a = a2 do
                                b/title
    ]
]
```

Combining values from multiple documents is not a problem for XML Query Algebra. Assume we have a second data source at *www.amazon.com/reviews.xml* that contains book reviews and prices, with the following DTD:

```
<!ELEMENT reviews (entry*)>
<!ELEMENT entry   (title, price, review)>
<!ELEMENT title   (#PCDATA)>
<!ELEMENT price   (#PCDATA)>
<!ELEMENT review  (#PCDATA)>
```

Consider a query that lists all books with their prices from both sources as an example. This query can be expressed in XML Query Algebra by:

```
Let bib0 = ("www.xml.com/bib.xml")/bib
Let review0 = ("www.amazon.com/reviews.xml")/reviews
books-with-prices [
    for b in bib0/book do
            for r in review0/entry do
            where b/title/data() = r/title/data() do
            book-with-prices [
                    b/title,
                    price-amazon [r/price/data()],
                    price-xml [b/price/data()]
            ]
]
```

In this case, only one more *let* expression has to be used to define the other new data source. All other parts are similar to previous queries. Because we only want to get the price value and not the *price* element, we have to use *data()* keyword to return the atomic data.

Order related operations, including indexing, sorting, and so on, are other kinds of special functions supplied by XML Query Algebra. (As discussed in Section 3.2.3, Xing does not supply them.) One such example is to return each book with its title and the first two authors, and an *<et-al/>* element if there are more than two authors. A possible XML Query Algebra solution is:

```
Let bib0 = ("www.xml.com/bib.xml")/bib
bib [
   for b in bib0/book do
         book [
               b/title,
               for p in index(b/author) do
               Where (p/fst/data() <= 2) do
                     p/snd/deref(),
               Where (p/fst/data() = 3) do
                     <et-al/>
         ]
]
```

Here *index()* is another built-in function in XML Query Algebra, which uses reference in order to preserve node identity when accessing local order. In the result pairs of the *index()* function, *fst* expresses the order index, and *snd* expresses a reference to result element. The built-in function *deref()* can de-reference an object. For detailed explanations and examples of *index()*, see Section 2.12 of [2].

The XML Query Algebra expression of Query 3 in Section 4.3 selected all titles of books published by Addison-Wesley after 1991. In that example, output order was not specified. Here we go back and show how to modify the query so that the titles are listed alphabetically.

```
Let bib0 = ("www.xml.com/bib.xml")/bib
bib [
for b in bib0/book do
         where b/publisher/data() = "Addison-Wesley" do
         where b/@year/data() > 1991 do
               sort t in (book [b/@year, b/title]) by t/title
]
```

XML Query Algebra provides a *sort* function to sort a forest. The syntax is: *sort Var in Exp1 by Exp2*. The variable *Var* ranges over all the items in the forest *Exp1*, and sorts them by the key value *Exp2*.

Chapter 5: Design of The Prototype

The Xing system is implemented in Java 2. It is portable on Win32 and different Unix platforms. It can run as a standalone application or as an applet from browsers. In this chapter, the main design issues are discussed, including the GUI design, the translation from Xing to XML Query Algebra, data structures, and algorithms.

5.1 GUI Design

Usability is a very important issue for end user systems. Therefore, the visualization of XML data is kept simple yet it still reflects the structure of the data. In Xing, a forms-like representation was chosen.

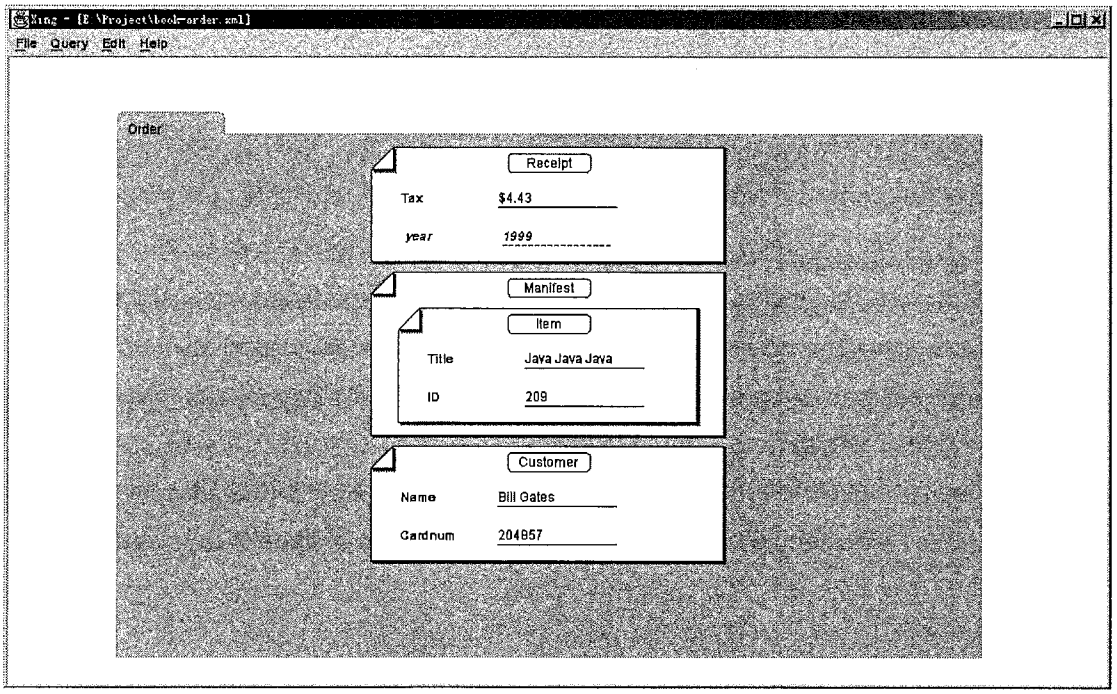

Figure 5.1. A sample representation for the structure of XML data

Figure 5.1 shows an example of how Xing represents the structure of an XML data source. We use a folder icon to represent an XML document, whose root node is the folder's name. An XML document consists of different categories of structured elements (that is, the elements that have subelements or attributes as children), which are all represented by document icons. The title of a document icon is given by the tag name of the structured element, and inside the icon there are different descriptions for plain elements (which have no children) and attributes. We use "**name:** value" to abbreviate the structure of plain elements. Attributes are also written this way. Two different base lines are used to distinguish elements from attributes.

By just using document patterns to express queries, users can extract information from XML data. By adding a second pattern as a result pattern, users can create document rules to get more control over the result.

To achieve a nice-looking GUI design, we have to consider different possible operations on the interface. One important issue is that the interface should be flexible enough for users to edit according to their needs. Then how should the icon size changes after users insert or delete a component inside or outside that icon? First, since the folder is at the uppermost level, we do not want to change its size anytime. Whenever it holds too many children to display, a scroll bar should show up to enable users to adjust the view area. Second, as the picture shows, document icons may be nested. We always maintain a minimum width and height for the innermost icon and adjust the size of the icons enclosing it. Only the icons enclosing the inserted or deleted component need to adjust their height. However, in addition to the enclosing icons, whenever a new document icon is inserted or deleted, we must also consider the changes to other nonrelated document icons to keep the same width for icons on the same level. More details will be discussed in Section 5.4.

5.2 Translating Xing to XML Query Algebra

To execute the queries composed by users, we have to translate them into a textual query language. This textual query is then executed by an XML database system. We can then link the XML database system to our system and supply a function to run queries directly.

As discussed in Section 2.1.1, we have chosen XML Query Algebra as the mid-layer language. Now the key point is how to do the translation. Because the described data

transformations may involve very complex restructurings, the formal semantics of rules gets quite complicated.

In this project, we only implemented the translation for document patterns, which is easier than the translation of document rules.

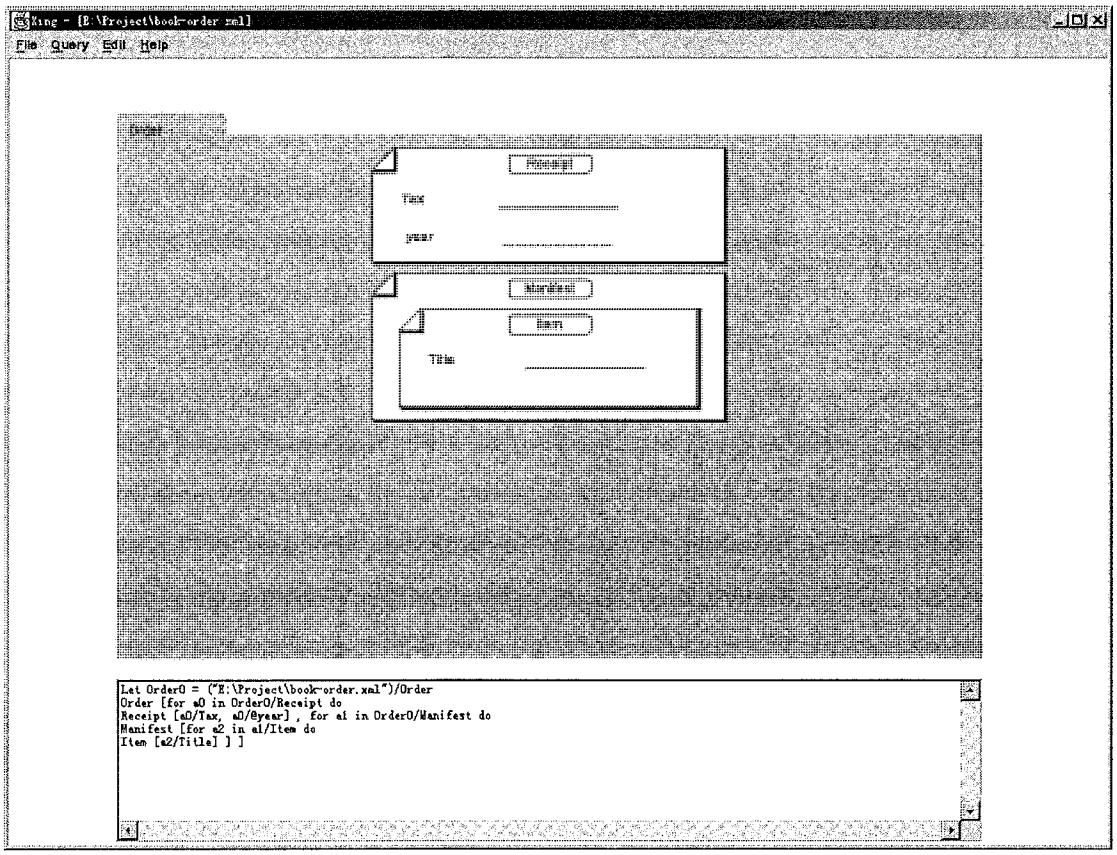

Figure 5.2. A query translation example

Figure 5.2 shows an example of a query translation. In addition to the pattern defining data projections, a text area shows up and displays the translated code in XML Query Algebra after the menu item *Generate Query* is selected.

The algorithm used to implement the translation will be discussed in Section 5.4.

5.3 Data Structure Design

To express the different components in the interface, in our program different classes are designed for different kinds of components. The whole structure of the query is treated as a tree. A folder icon is the root, which can hold a variable number of document icons as children. The children of document icons can be document icons, plain elements, or attributes.

There are different classes for folder icon, document icon, plain element and attribute, each of which has different properties to represent their features.

First, every document icon has width and height properties, and they are flexible to adjust automatically during the query editing. Each class has two width variables: *maxWidth* and *minWidth*. The current width in which an icon should be displayed is *maxWidth*, while *minWidth* is the minimum width an icon must maintain. The reason why they can be different in general is because we always display the same width (*maxWidth*) for icons in the same level and the variable *maxWidth* of any icon may be changed in case any icon insertion or deletion happens on the interface. To decide how to change the value of *maxWidth*, the value of *minWidth* has to be considered. In Section 5.4, this process will be explained in more detail.

Second, every icon class, either folder or document, has an array variable *elArray* to record its child components. That member array must be able to hold different types of objects, since the possible child class types may include structured element class types, plain element class types, or attribute class types.

A plain element or attribute consists of a name field and a condition field. The name field, always in bold font, represents the node name. The condition field represents a value or condition that constrains the elements being queried.

Finally, our program uses an XML parser [20] to analyze the selected XML data and generate an object to access its content. Moreover, if the XML document specifies its document format with a DTD, we use a DTD parser [21] to analyze it for generating a pop-up menu.

5.4 Algorithms

5.4.1 Dynamic Icon Resizing

It is easy to decide the height of a document icon after any offspring component is added or deleted, since its siblings' height will not be affected. However, it is complicated to adjust the width of document icons because we must keep their siblings' width consistent.

Figure 5.3 shows an example of how the changes can occur. Under the root folder there are two icons called *Receipt* and *Manifest*. In the top figure, a plain element and an attribute are inserted into the *Receipt* icon. The insertion causes that the height of *Receipt* is increased. However, the insertion of plain elements or attributes does not affect the width of their parent icon. In the bottom figure, a new icon *Item* is inserted into the *Manifest* icon. As the figure shows, the *Manifest* icon enlarges to hold its children. Meanwhile, the *Receipt* icon enlarges, too, to keep the same width as *Manifest* because they are on the same level. There is also a minimum width that must be maintained by the *Item* icon, since it is the innermost. So, how do we set the width of *Manifest* after *Receipt* is removed, or the width of *Receipt* after *Manifest or Item* is deleted? To solve that, there are two variables *maxWidth* and *minWidth* defined in a document icon.

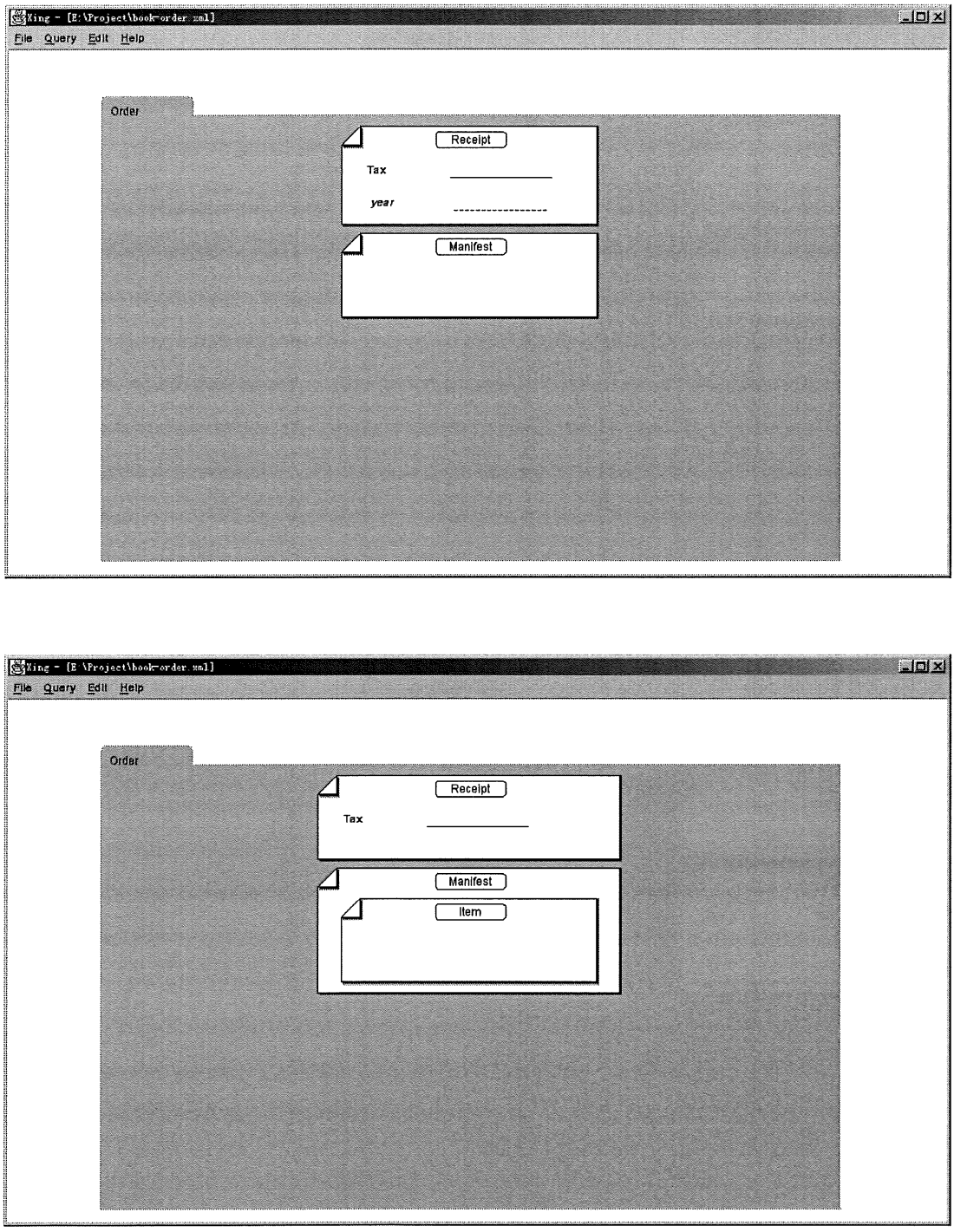

Figure 5.3. Change size after a new component added

Since the height adjustment is easy to implement, this paper omits the algorithm about the size handling after a plain element or an attribute is added or deleted. Below is the algorithm description in java-like pseudo-code about how to handle the size changes after an icon is added. The first method *addStructElement()* is a member method of the document icon class, called when a child icon is added to the current icon. It initializes the new child icon's size, then calls the *bottomUp()* method of the current icon to recursively update the size of its ancestors upwardly. The constant *WIDTH* is the minimum width that an icon must hold. The constant *HEIGHT* is the minimum height that an icon must hold. The constant *SPACE* is the display space between the child and its parent.

```
addStructElement(String arg) {
    create a new  icon object: tmp; //whose minWidth is always WIDTH as default
                                    //whose height is always HEIGHT as default
    set tmp's title as arg;
    insert the new icon into the current icon container as a child;
    if (the current icon is root) {
            if (the current icon has no children except tmp)
                    set tmp.maxWidth = WIDTH;
            else
                    set tmp.maxWidth = maxWidth of the other siblings
    } else {
            if (maxWidth of the current icon equals the basic value WIDTH)
                    set tmp.maxWidth = WIDTH;
            else
                    set tmp.maxWidth = maxWidth of the other siblings
                    and we know that the value must be maxWidth - SPACE;
            bottomUp(tmp);
    }
}
```

After an icon is added, the program continues to update the properties of its ancestors from bottom to top. The parameter *s* of the following method is the updated child of the current icon.

```
bottomUp(a document icon: s) {
        if (the current icon is root) {
            declare q as document icon class type;
            go through every child of the current icon and refer to it by q {
                        set q.maxWidth = s.maxWidth;
                        q.topDown();
            }
        } else {
            if (maxWidth of the current icon equals s.maxWidth) {
                        enlarge maxWidth of the current icon to s.maxWidth+SPACE;
                        enlarge minWidth of the current icon to s.minWidth+SPACE;
                        update the height of current icon;
                        topDown();
                        call bottomUp(this) of the parent icon;
            } else {
                        //the current icon has bigger width than s, so no width to be enlarged
                        update the height of current icon;
                        if ( the current icon's parent is not root )
                                    call bottomUp(this) of the parent icon;
                        //otherwise, the recursive update stops
            }
        }
}
```

The following method updates the width of the siblings, those that are not direct ancestors of the inserted icon, from top to bottom. The children of the current icon may be document icons (structured elements), plain elements, or attributes. The displayed width (*maxWidth*) of the child icons is always set to a constant value, *SPACE*, lower than the displayed width of the parent icon.

```
topDown() {
    declare s as the document icon type;
    go through every child of the current icon and refer to it by s {
            if ( s is document icon type ) {
                    set s.maxWidth = maxWidth-SPACE;
                    call topDown() of the object s;
            }
    }
}
```

Below is a description of the algorithm for how to handle the size changes after an icon is deleted. The following method is a member method of the document icon class, called after one of its child icons is deleted. This method is used to update the properties of its ancestors from bottom to top.

```
void bottomUpRev() {
        int width;
        if (the current icon does not have any child of icon type )
            set width = WIDTH;
        else {
            look for the icon that holds the maximum value of minWidth among the children
            of the current icon, and store the value in a variable max;
            set width = max+SPACE;
        }
        if (the current icon is root) {
            declare s as document icon type;
            go through every child of the current icon and refer to it by s {
                        set s.maxWidth = width-SPACE;
                        s.topDown();
            }
        } else {
            set maxWidth = width;
            set minWidth = width;
            update the height of current icon;
            topDown();
            call bottomUpRev() of the parent icon;
        }
}
```

5.4.2 Generation of Pop-up Menus

When a user clicks the right mouse button on an icon, an icon's title field, or the name field of a plain element or an attribute, a pop-up menu shows up. Only those items that can be added on the icon or the field are listed on the menu. How can we decide which items to show in the pop-up menu? For example, using the DTD description in Section 3.2, if a user clicks on the icon with title *article*, only *title, author,* and *editor* are listed as menu items. On the other hand, if a user clicks on a child icon of root folder with empty title or title '*' (the wildcard), then all the child nodes of both *article* and *book* should be listed as menu items.

Below is the algorithm to compose the dynamic pop-up menu. Actually, there is no big difference between the algorithm handling the clicks on a title or name field and the algorithm handling the clicks on a folder or an icon. Here we just present the algorithm that handles the clicks on an icon's title field. In case an icon is clicked on, the program just needs to go through all of the icon's children and generates a menu item for each of them. When the name field of a plain element or an attribute is clicked on, the program just needs an additional check on the icon's children and only generate menu items for those whose type match with the focused component.

The following code segment does those initialization works. The variable *focusOwner* is the icon users are clicking on. The variable *dtd* is an object that represents the DTD tree structure of the selected XML data.

```
get root element of dtd and assign it to the variable root;
declare an array called elements, used to hold selected objects;
test every child node of root, add it to elements if the node is a structured element;
declare and initialize a popup-menu, called popup;
declare icons as an array of icons;
scan from the parent icon of focusOwner until root, insert every icon into icons in order;
```

The following part searches appropriate items to compose the pop-up menu. It uses a variable *notFound* as the flag to record whether there is any matched item found. Whenever there is no node name that matches with an icon's name in any level, the search stops.

```
boolean notFound = false;
during notFound is still false, go through every member of icons from root to children, and
refer to it by a variable cur {
        notFound = true;
        declare a temporary array variable, called tmpArray;
        if ( the title of cur is empty or '*' ) {
            notFound = false;
            test every child node of the array member of elements, add it to
            tmpArray if the node is a structured element;
            elements = tmpArray;
        } else {
            go through every member of elements, and refer it to a variable elmt {
                if (elmt has a tag same to the title of cur) {
                        notFound = false;
                        test every child node of elmt, add it to tmpArray if the node is a
                        structured element;
                        elements = tmpArray;
                }
            }
        }
}
go through every member of elements, create a menu item with its name, add it to popup.
show the pop-up menu on focusOwner;
```

5.4.3 Translation of Xing Queries

To do the translation, the algorithm treats the UI components as a tree structure and goes through every node of the tree in the depth-first order. Thus, there is a separate queryGenerate() method in the main class, the folder icon class, and the document icon class. The program generates an abstract syntax tree (AST) to express the structure of the translated XML Query Algebra query. In the current version, only pattern queries can be translated. The translation does not allow the tag and the content of an element or an attribute to be empty at the same time.

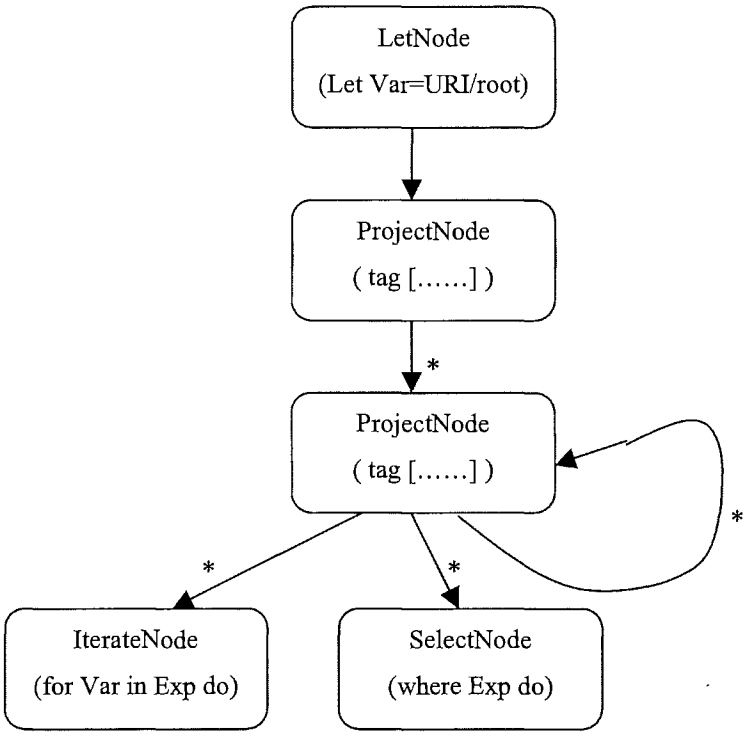

Figure 5.4 AST of the query translation

As the above figure shows, the AST consists of several node types to represent different kinds of query statements, which are introduced in Chapter 4. *LetNode* represents the *let* expression to bind a variable to an XML data source. *ProjectNode* represents the projection operation to construct a new document. *IterateNode* represents the *for* expression to iterate over all subelements of one element. *SelectNode* represents the *where* expression to select values that satisfy the predicates. Every node has a *print()* method to output the query that it represents. *LetNode* is the root of the AST, whose child is a *ProjectNode* to construct the root of the new document. Every *ProjectNode* may have multiple children including *ProjectNode, IterateNode,* and *SelectNode,* because the projection operation can be nested. Because only document icons can be added as children into the folder icon, the *ProjectNode* mapping to the folder icon has only children of type *ProjectNode.*

In the main class of the program, the following actions are executed after a user selects the menu item "Generate Query". The global variable *ast* is used to represent the generated XML Query Algebra AST, whose root node is of type *LetNode.* The variable *root* is

the string name of the XML document's root node. The variable *xmlFile* is the URL or string that represents the position of the XML document. The variable *folderPane* is the object that represents the root folder icon of the Xing query. The member method *queryGenerate()* of *folderPane* returns a node of *ProjectNode* type, which is inserted into the AST.

```
ast = new LetNode(root, xmlFile, root+"0");
if (folderPane exists) {
    call folderPane.queryGenerate() and add the return node to ast as a child node;
}
```

Below is the method queryGenerate() in the folder icon class. The variable *titleName* is the title of the folder icon, that is the name of the root node. The string *titleName+"0"* is the variable that is bound to the root node by the *let* expression.

```
ProjectNode queryGenerate () {
        ProjectNode folder = new ProjectNode(titleName, titleName+"0");
        go through every child icon of root, call its method queryGenerate(titleName+"0") ,
        and add the return node to folder as a child node;
        return folder;
}
```

Below is the method queryGenerate() in the document icon class, which also returns a node of type *ProjectNode*. The parameter *root* is the variable that binds to the root node in the *let* expression. If a plain element or an attribute's condition field is empty, only a projection node is created as a child of the returned node; otherwise, both a selection node and a projection node are created as children of the returned node. The variable *index* is a static integer with initial value 0 in the document icon class, and is used as the index of a variable *a#* that is bound to the current node. Every time this method is called, *index* is increased by one to ensure that every node binds with a different variable *a#*.

```
queryGenerate (root) {
        String titleName;
        if (the icon's title field is empty or wildcard)
            set titleName = "*";
        else
            set titleName = the text in title filed;
        ProjectNode icon = new ProjectNode(titleName, root+"/"+titleName);
        if (the document icon has no any children) {
        //Which means that icon has no any children. The query should get the whole node
            return icon;
        }
        int j=index++; //the index of current variable a#, which binds to the current node
        create a new object IterateNode("a"+j, root+"/"+titleName)) and add it to icon as a
        child node;
         //generate selection part of the query
        go through every child of the current icon. If that is a plain element or an attribute
        and its condition field is not empty, create a new object of SelectNode type and add
        it to icon as a child node;
        //generate projection part of the query
        go through every child of the current icon. If that is a plain element or an attribute,
        create a new object of ProjectNode type and add it to icon as a child node . If that
        is a document icon, do a recursive call on its member method
        queryGenerate(root+"/"+titleName) and add the return node to icon as a child
        node;
}
```

Chapter 6: Conclusion and Future Work

Xing is trying to supply a very direct way for end users to query XML databases, and we believe it is successful in achieving this goal. Two main reasons to support our belief are: 1. The document metaphor used in Xing is based on a common form concept, which is an easy solution for naïve users. 2. It frees users from understanding complex textual query languages.

One trade-off, usually existing in data query systems, is between expressive power and ease-of-use. Xing also tries to be able to express more complicated queries and supply users more query functions. Although its expressiveness is limited compared with some textual query languages, like XML Query Algebra, it seems to be generally sufficient for non-professionals to pose simple queries conveniently. Meanwhile, it saves users lots of energy and reduces lots of pain by giving a possibility to avoid studying complex languages. Basically, from the need in the real world we can see a pretty promising prospect for the use of a visual and intuitive programming interface to query XML-based web documents.

One interesting feature of our Xing implementation is that it can be run as either a standalone application or an applet from browsers. As an applet, it can be open to the public through the Internet for trial and getting comments/feedback. Another feature is the pop-up menu assistance for users through analyzing the DTD of the XML data worked on. It gives further clues and tips during a user's query design, and avoids that users specify any meaningless names or make typing errors.

Meanwhile, there are still lots of issues left on Xing's future research. One aspect is to define a formal semantics for Xing because it is the bridge between end user graphical or textual notations and the real meaning of query expressions. No matter whether to do query translation or to compile and execute the query, it is always a key step to understand the semantics. Actually, some work on the semantics has been done and discussed in [1]. With semantics, it also becomes possible to determine the exact expressive power of Xing, which will help users to utilize the advantage of Xing system better.

Apart from the theory issues, one practical future work is to implement the rules translation. Moreover, we may develop a new compiler or use any existing compiler and embed it to Xing system so that users can execute the queries directly in Xing environment.

Again, the semantics will help a lot on that. Another possible improvement on the system is to extract DTD information from pure XML data. Currently, the system can supply pop-up menu assistance only if there is an explicit DTD in the document. However, it is also possible to analyze a document and extract its DTD structure by Xing itself. One such tool is XTRACT [14]. Since plenty of existing XML documents do not have DTD specification, which are only pure data or specified by schema, that function can enable Xing's nice feature to be exploited in a much broader range.

Bibliography

[1] M. Erwig. Xing: A Visual Language for XML. *16th IEEE Symp. on Visual Languages* (VL 2000), 47-54, 2000.

[2] P. Fankhauser, M. Fernández, A. Malhotra, M. Rys, J. Siméon, and P. Wadler, editors. The XML Query Algebra. *W3C Working Draft,* 15 February 2001.

[3] M. M. Burnett. Visual Programming. In Webster, J. G., editor, *Encyclopedia of Electrical and Electronics Engineering.* John Wiley & Sons, 1999.

[4] M. Erwig. XML Queries and Transformations for End Users. XML 2000, 259-269, 2000.

[5] A. Deutsch, M. Fernández, D. Florescu, A.Levy, and D. Suciu. A query language for XML. *International World Wide Web Conference*, 1999.

[6] D. Chamberlin, D. Florescu, J. Robie, J. Siméon, and M. Stefanescu, editors. XQuery: A Query Language for XML. *W3C Working Draft,* 15 February 2001.

[7] M. Fernández, J. Simeon, P. Wadler. XML Query Languages: Experiences and Exemplars.

[8] S. Ceri, S. Comai, E. Damiani, P. Fraternali, S. Paraboschi, and L. Tanca. XML-GL: A Graphical Language for Querying and Restructuring XML Documents. In *8th Int. World Wide Web Conference,* 1999.

[9] S. Cohen, Y. Kanza, Y. Kogan, W. Nutt, Y. Sagiv, and A. Serebrenik. EquiX – Easy Querying in XML Databases. In *2nd ACM SIGMOD Int. Workshop on The Web and Databases,* pages 43-48, 1999.

[10] T. Bray, J. Paoli, C. M. Sperberg - McQueen, and E. Maler, editors. *Extensible Markup Language (XML) 1.0,* 2000. http://www.w3.org/TR/REC-xml.

[11] D. Chamberlin, P. Fankhauser, M. Marchiori, and J. Robie, editors. XML Query Requirements. *W3C Working Draft,* 15 February 2001.

[12] M. Fernández and J. Robie, editors. XML Query Data Model. *W3C Working Draft,* 15 February 2001.

[13] D. Chamberlin, P. Fankhauser, M. Marchiori, and J. Robie, editors. XML Query Use Cases. *W3C Working Draft,* 15 February 2001.

[14] M. N. Garofalakis, A. Gionis, R. Rastogi, S. Seshadri, K. Shim. XTRACT: A System for Extracting Document Type Descriptors from XML Documents. In *Proceedings of ACM SIGMOD Conference on Management of Data,* pages 165--176, May 2000.

[15] M. Marchiori, editor. *XML Query*, 2000. http://www.w3.org/XML/Query.html.

[16] M. Fernández, J. Simeon, P. Wadler. An Algebra for XML Query. In *Int. Conf. on Foundation of Software Technology and Theoretical Computer Science,* LNCS 1974, 2000.

[17] J. Cowan, editor. *XML Information Set,* 2000. http://www.w3.org/TR/xml-infoset.

[18] D. Chamberlin, J. Robie, and D. Florescu. Quilt: An XML Query Language for Heterogeneous Data Sources. In *3nd ACM SIGMOD Int. Workshop on The Web and Databases*, 2000.

[19] J. Clark and S. DeRose, editors, *XML Path Language (XPath), Version 1.0,* 1999. http://www.w3.org/TR/xpath.

[20] *Java™ APIs for XML Processing (JAXP)*. http://java.sun.com/xml/xml_jaxp.html.

[21] Ronald Bourret. *DTD Parser and Schema Converters.* http://www.rpbourret.com/schemas/index.htm.