XML is a very versatile data format that has been used to represent many different kinds of data, including web pages, books, business and accounting data, programming interfaces, vector graphics, system logs, and games. In a short span of time, it has gained wide acceptance as the document and data standard on the web. As more and more XML data gets generated everyday, a lot of research focus has been on query languages for XML. The World-Wide Web Consortium (W3C) has chosen XQuery as the standard language for querying XML. From an end-user point of view, XQuery sacrifices usability for expressiveness.

We introduce FoXQ, a visual language that enables end users to query XML. FoXQ brings a lot of the functionality of XQuery within the reach of the end users without getting them embroiled in the intricacies of XQuery syntax. The query interface is form-based and the query model is based on a document metaphor in which the users formulate queries by filling out forms.
FoXQ: A Visual Query Language for XML

by

Robin Abraham

A THESIS

submitted to

Oregon State University

in partial fulfillment of
the requirements for the
degree of

Master of Science

Completed September 26, 2003
Commencement June 2004
ACKNOWLEDGMENTS

I would like to thank Dr. Martin Erwig without whose patient guidance this work would not have been possible. I would also like to thank the other members of my committee, Dr. Jon Herlocker, Dr. Margaret Burnett, and Dr. Rod Harter.

I would also like to thank to my parents and my brother for their encouragement and support.
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Chapter</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chapter 1</td>
<td>Introduction</td>
<td>1</td>
</tr>
<tr>
<td>1.1</td>
<td>Why XML?</td>
<td>1</td>
</tr>
<tr>
<td>1.2</td>
<td>Querying the Web</td>
<td>2</td>
</tr>
<tr>
<td>Chapter 2</td>
<td>Literature Review</td>
<td>6</td>
</tr>
<tr>
<td>Chapter 3</td>
<td>The FoXQ Interface</td>
<td>10</td>
</tr>
<tr>
<td>3.1</td>
<td>The Document Metaphor</td>
<td>10</td>
</tr>
<tr>
<td>3.2</td>
<td>The Design of the FoXQ Interface</td>
<td>12</td>
</tr>
<tr>
<td>3.3</td>
<td>Constructing Queries in FoXQ</td>
<td>16</td>
</tr>
<tr>
<td>3.4</td>
<td>Viewing Results of FoXQ Query Execution</td>
<td>18</td>
</tr>
<tr>
<td>3.5</td>
<td>FoXQ Examples</td>
<td>20</td>
</tr>
<tr>
<td>3.5.1</td>
<td>Selection Queries</td>
<td>20</td>
</tr>
<tr>
<td>3.5.2</td>
<td>Projection Queries</td>
<td>22</td>
</tr>
<tr>
<td>3.5.3</td>
<td>Restructuring Queries</td>
<td>25</td>
</tr>
<tr>
<td>3.6</td>
<td>Future Work</td>
<td>29</td>
</tr>
<tr>
<td>3.6.1</td>
<td>Support for XML Attributes and Empty Elements</td>
<td>30</td>
</tr>
<tr>
<td>3.6.2</td>
<td>Support for XQuery Functions</td>
<td>31</td>
</tr>
<tr>
<td>3.6.3</td>
<td>DTD-Based Help in the Query Builder</td>
<td>31</td>
</tr>
<tr>
<td>3.6.4</td>
<td>Context-Sensitive Help in the Query Builder</td>
<td>32</td>
</tr>
<tr>
<td>3.6.5</td>
<td>Support for Flattening Queries</td>
<td>32</td>
</tr>
<tr>
<td>3.6.6</td>
<td>Usability Testing</td>
<td>33</td>
</tr>
<tr>
<td>Chapter 4</td>
<td>Translation from FoXQ to XQuery</td>
<td>34</td>
</tr>
<tr>
<td>4.1</td>
<td>Inferring Regrouping Operations</td>
<td>35</td>
</tr>
<tr>
<td>Chapter</td>
<td>Section</td>
<td>Page</td>
</tr>
<tr>
<td>-----------</td>
<td>-------------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>4.2</td>
<td>Translation of Selection and Projection Queries</td>
<td>37</td>
</tr>
<tr>
<td>4.3</td>
<td>Translation of Restructuring Queries</td>
<td>39</td>
</tr>
<tr>
<td>Chapter 5</td>
<td>System Implementation</td>
<td>42</td>
</tr>
<tr>
<td>5.1</td>
<td>System Architecture</td>
<td>42</td>
</tr>
<tr>
<td>5.2</td>
<td>The FoXQ Interface</td>
<td>43</td>
</tr>
<tr>
<td>5.2.1</td>
<td>The Query-Builder Component</td>
<td>44</td>
</tr>
<tr>
<td>5.2.2</td>
<td>The Result-Display Component</td>
<td>45</td>
</tr>
<tr>
<td>5.3</td>
<td>The Translation Module</td>
<td>46</td>
</tr>
<tr>
<td>5.3.1</td>
<td>Inferring Regrouping Operations</td>
<td>49</td>
</tr>
<tr>
<td>5.3.2</td>
<td>Translation of Selection and Projection Queries</td>
<td>50</td>
</tr>
<tr>
<td>5.3.3</td>
<td>Translation of Restructuring Queries</td>
<td>53</td>
</tr>
<tr>
<td>5.4</td>
<td>File Format for FoXQ Queries and Results</td>
<td>56</td>
</tr>
<tr>
<td>Chapter 6</td>
<td>Conclusions</td>
<td>60</td>
</tr>
<tr>
<td>Bibliography</td>
<td>62</td>
<td></td>
</tr>
</tbody>
</table>
# LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>Document Icons</td>
<td>10</td>
</tr>
<tr>
<td>3.2</td>
<td>The FoXQ interface</td>
<td>17</td>
</tr>
<tr>
<td>3.3</td>
<td>The result-viewer interface of FoXQ</td>
<td>19</td>
</tr>
<tr>
<td>3.4</td>
<td>List of books in bibliography</td>
<td>21</td>
</tr>
<tr>
<td>3.5</td>
<td>Author and title information from books</td>
<td>22</td>
</tr>
<tr>
<td>3.6</td>
<td>Titles of books published by Addison-Wesley in 1991</td>
<td>23</td>
</tr>
<tr>
<td>3.7</td>
<td>Books from a particular year</td>
<td>24</td>
</tr>
<tr>
<td>3.8</td>
<td>Renaming of “book” tag to “result”</td>
<td>26</td>
</tr>
<tr>
<td>3.9</td>
<td>Book titles grouped by authors</td>
<td>27</td>
</tr>
<tr>
<td>3.10</td>
<td>List of titles found both on amazon.com and bn.com</td>
<td>28</td>
</tr>
<tr>
<td>4.1</td>
<td>Variable/node bindings for a selection/projection query</td>
<td>35</td>
</tr>
<tr>
<td>4.2</td>
<td>Variable/node bindings for a restructuring query</td>
<td>36</td>
</tr>
<tr>
<td>5.1</td>
<td>FoXQ implementation architecture</td>
<td>42</td>
</tr>
</tbody>
</table>
# LIST OF TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>FoXQ icons</td>
<td>11</td>
</tr>
<tr>
<td>5.1</td>
<td>Format for saving FoXQ queries to file.</td>
<td>58</td>
</tr>
<tr>
<td>5.2</td>
<td>Format for saving FoXQ results to file.</td>
<td>59</td>
</tr>
</tbody>
</table>
CHAPTER 1

INTRODUCTION

1.1 Why XML?

Organizations running applications on multiple platforms have been faced with the daunting task of replicating their data and applications on the different platforms because of problems with interoperability. Java solved part of that problem with its "write once, run anywhere" philosophy. Even so, Java's portability has been seriously compromised as proprietary data formats have been used for years, enabling an application to run on multiple platforms, but not across businesses in a standardized way. This is part of the promise of XML — full data interoperability. Some of the uses XML has been put to, are as follows [43].

1. For presentation Increasingly, XML is being used for presentation — the content is represented in XML while XSL and XSLT are used to control the presentation.

2. For communication

- XML can be used to transfer data between applications.
- XML-RPC is concerned with communication between components within an application, or to a shared set of services functioning across applications.

- XML is also being used for data transfer between businesses.

3. For configuration  More and more application developers are turning to XML for storing system configuration and user-profile information as XML data. This open format makes it easier to transfer this information between different applications or between different versions of the same application.

1.2 Querying the Web

The world-wide web has grown into one of the largest information resources. At the same time it is very difficult to search for information on the web. The main reasons for this situation are (i) the fact that much of the data is available in unstructured or semistructured form and (ii) the lack of expressive and easy-to-use query languages to locate relevant information. The prevailing approach to querying the web is to use search engines. The majority of computer users, who do not have any formal computer science education, use simple keyword-based searches with search engines. In many cases the huge number of returned links makes it very difficult to identify the most relevant information sources. Some search engines provide logical operators like AND and OR, but there is strong empirical evidence that end users cannot effectively use them [46]. Although it is possible in some cases to improve the precision of keyword queries through the automatic insertion of logical operators [23], the problem of too many results principally remains.
The adoption of XML as a new standard to store data on the web promises new opportunities to express more precise queries since XML provides structure to data that can be exploited by appropriately defined query languages. However, although there exist quite a few proposals for XML query languages, for example, [50, 16, 10, 32, 34, 17, 26] to mention just a few, these are generally usable only for users with profound database expertise, which most end users typically lack.

As XML gets adopted as the document and data standard on the web, we are faced with a shortage of tools that allow us to easily access this data. The problem is compounded when we look at it from the perspective of different classes of users, such as database experts or end users who have no prior programming experience. The level of functionality or sophistication the different classes of users expect from their access tools is different. For example, a database expert would expect greater ability to manipulate the XML data whereas an end user would expect greater usability.

At present, users can query the web using any one of the following methods:

1. By using search engines. The main problem with this approach is that search engines typically return too many results since they do not exploit the structure provided by XML. More intelligent searches could be formulated if search engines offered querying facilities to exploit the XML structure.

2. By using dedicated database front ends for a particular data set. In this case, we sacrifice flexibility for accuracy. First of all, queries are limited to a small selected subset of data resources. Moreover, every time the schema of the XML data is changed, the program that serves as the front end has
to be changed accordingly. That makes this approach time consuming and uneconomical from a maintenance point of view.

3. By using specific XML query languages like XQuery. This method is very flexible and powerful since it makes full use of the structure of the XML data. The only problem with this method is that it cannot be adopted by end users who might have little or no programming experience at all.

Our approach is to pursue the third option and make it accessible to end users by an easy-to-use visual query front end. Our goal is not to invent a visualization for every part of XQuery, which would compromise the goal of simplicity. Instead, we make a significant part of XQuery usable by end users, thereby making the power of XQuery accessible to a large number of internet users. This approach requires careful consideration of what end users are capable of doing and what they cannot do. We believe that to be successful, research in end-user programming has to be taken into account.

We have designed a system, called FOXQ (Form-based XML Querying). This system allows the users to design their own forms, which are then used to extract data from XML databases. Our work builds on previous work that introduced a visual notation that allows the formulation of queries by patterns and rules [20, 21]. A formal semantics for the resulting visual query notation has also been proposed [22]. Based on that work we have designed a visual query interface that allows the formulation of selections, projections, and grouping queries using forms. We have implemented a compiler that translates such form-based queries into XQuery. The visual interface is implemented in Java and can be installed and used as a stand-alone program or as an applet. The translated queries can be executed on any XQuery implementation. This approach allows
the easy installation and widespread use of the query interface on almost any computer.

The promotion of the visual query interface can have positive effects on the XML/database development: On the one hand, a widespread use of the user interface creates additional demand for XQuery implementations and fosters the database research in many related XML areas (query optimization, indexing, and so on). On the other hand, an increased use of the query interface urges information providers to make their information available in XML.
CHAPTER 2
LITERATURE REVIEW

The XML query working group [42] has defined query requirements for XML [8] (including a great number of different example applications and queries). Together with their definition of an XML query data model [25] and an XML Query Algebra [26, 17, 27] the requirements comprise a precise description of an XML query language [3]. The XML query data model is itself based on the XML information set [14] that provides a description of the information available in a well-formed XML document. The XML Query Algebra is intended to be a formal basis for XML query languages. The major parts are path expressions and a for construct that allows expression of iterations over sequences of elements. The XML Query Algebra proposal presents an elegant and powerful core query system: the combination of conditional expressions with the ability to rather arbitrarily combine and nest different expressions allows selection, projection, join, quantification, and restructuring to be expressed. There is also support for sorting and aggregation. Quilt [9] is a functional language that builds directly on the XML Query Algebra and provides specialized syntax for several parts of the XML Query Algebra.

Research in visual languages and end-user programming has also created proposals for XML query languages. XML expressions are multi-way trees, and trees can be expressed as pictures. XML-GL [5] is a language proposal that is based on this idea. However, the trees represent rather an abstract visual
syntax of XML documents, which supports formal language manipulations well [19], but a pictorial representation does not alone necessarily imply usability by end users. In fact, XML-GL is not claimed to be aimed at end users; instead the focus is on formal properties of the presented notation.

Form-based interfaces for web-based data entry are widespread, and this use of the term "form based" may be what comes to mind. However, forms can also be used in a more general way, as the basis of a programming paradigm. The most well-known member of the form-based paradigm is the commercial spreadsheet, in which users start with a blank form (spreadsheet) and insert labels and formulas to specify its structure and content. The popularity of commercial spreadsheets provides strong practical evidence of the usability of the form-based paradigm by end users. Additional laboratory and empirical data abounds as to the usability of this paradigm (see [44] for a partial summary). A form-based query interface is provided by EquiX [11] whose most important goal is to achieve a simple interface. However, the form metaphor is only used half-heartedly on the outermost level, and nesting is expressed by simple indentation. The forms are generated semi-automatically, driven by a DTD, but this imposes a severe restriction since only data sources providing a DTD can be queried. Moreover, the expressiveness of EquiX is limited, and it is not possible to reformat the query results. It is also not possible to express conditions on arbitrarily deeply nested elements (which is possible in most textual XML query languages through path expressions or something similar). Such queries were termed deep queries in [4]. Similar constraints apply to QBE-like [62] languages for the nested relational data model (for example, [40, 58]). Although these allow restructuring of data, they too depend on the presence of a schema and even demand the display of the complete schema of queried rela-
tions. They are not able to express deep queries either. *DoodlE* [15] and *VQL* [57] are form-based visual query languages that are intended as general-purpose database languages. Both are rule-based languages, and both are similar to our approach in their visual appearance using nested rectangles for displaying data as well as queries. In *VQL*, but not in *DoodlE*, it is possible to pose queries about the schema. This is a very important feature, especially in cases when the user has limited or even no knowledge of the schema. Although not implemented yet, schema queries are principally possible in *FoxQ* through the use of end-user regular expressions as tags.

Besides the form-based paradigm, our approach includes use of the rule-based paradigm. This approach was pioneered for end-user programming in recent years by the two end-user rule-based languages, *AgentSheets* [49] and *KidSim/Cocoa* [51]. The usability of both of these languages has been studied empirically [48], and has been found to be not only viable, but enjoyable by end users.

Since the W3C adopted *XQuery* [12] as the standard query language for XML, there has been considerable work on the development of query engines for XQuery. Quite a few of these are available for download and use for research purposes. Of these, *Galax* [41], *Qexo* [31], *XQuench* [53], and *XQuery Lite* [54] are open-source. *XStreamDB* [13], *SQL/XML-IMDB* [47], and *X-Hive’s XQuery demo* [59] are all native XML databases with XQuery engines. Systems like BEA’s Liquid Data [2], Enosys Software’s XQuery engine [52], E-XMLMedia’s XMlizer [18], Oracle’s XML DB [45], Nimble Technology’s Nimble Integration Suite [56], and XML Global’s GoXML DB [30] use XQuery for extracting XML data for XML-RDBMS integration. Oracle’s XML DB uses SQL/XML, which is a blend of SQL and XPath, to “step into” XML data (XPath allows the user
to bind variables to arbitrary nodes in the XML data. These variables can then be used to access the data). Cerisent’s XQE [6], GAEL’s Derby [28], Ipedo’s XML database [35], IPSI’s IPSI-XQ engine [36], and Software AG’s QuiP [1] are all standalone XQuery engines. Fatdog’s XQEngine [24] is not a standalone engine and is intended to be embedded in Java applications. Cognetic Systems’ XQuantum system [55] is a research implementation of a subset of the XQuery specifications and the type inference system.

None of the XQuery engines mentioned in the previous paragraph are targeted at end users because:

- The XQuery editors, if available, are all basically textual. Some provide extra help by recognizing keywords and XQuery grammar, but they are not visual query languages per se.

- Quite a few of them represent XML data and DTDs as trees [60, 61]. This is unfortunate since the main advantage of a GUI is that it can decouple the interface from the underlying data representation. For end users, it is probably worse to have to edit abstract syntax than concrete syntax.

The FoXQ system, on the other hand, is targeted at end users. It requires minimal textual input from the users, and shields them from XQuery and XML syntax by using the *document metaphor* for the formulation of queries and for the display of XML data.
CHAPTER 3

THE FoXQ INTERFACE

3.1 The Document Metaphor

The general idea of expressing queries in FoXQ is to create query forms that are used as patterns to find matching elements in an XML database. Query forms are represented by so-called document icons, which are graphical representations of documents. Figure 3.1 shows a sample query. We observe that the query has two folder icons, each containing a document icon. We call such a query, with two folder icons, a document rule. The document icon on the left represents what is called a selection pattern and the document icon on the right represents the so-called projection pattern. The user can specify the selection conditions in the selection pattern by filling in values like in a form. The selection pattern

![Diagram of Document Icons]

FIGURE 3.1: Document Icons.
is used to specify the data to be extracted, and the projection pattern controls the way the extracted data is to be formatted. If the user does not need to do any formatting of data, she might use only one pattern, in which case, the same pattern would be used for selection and projection.

The visual elements of FoXQ and their correspondence to XML is summarized in Table 3.1. *Elements* are simply notational abbreviations of document icons – the element name (the text area on the left) corresponds to the name of the document icon and the element content (the text area on the right) corresponds to the content within the document icon. We have adopted the simplified notation using elements to conserve screen real-estate. The document metaphor has the advantage that the user does not have to concern herself with variables and the nodes of the XML document tree.

<table>
<thead>
<tr>
<th>FoXQ icons</th>
<th>XML representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Folder icon</td>
<td>XML root element</td>
</tr>
<tr>
<td>Document icon</td>
<td>Pattern that matches a set of XML elements</td>
</tr>
<tr>
<td>Elements</td>
<td></td>
</tr>
<tr>
<td>Document Rule</td>
<td>XML query</td>
</tr>
</tbody>
</table>
3.2 The Design of the FoXQ Interface

We have drawn heavily from HCI research in the design of the FoXQ interface. The interface design has also been influenced to a great extent by the work on DENIM and SILK interfaces [38, 39].

The Cognitive Dimensions of Notations (CD) framework [33] is an excellent yardstick for judging the usability of the FoXQ interface. Since we have not actually done any end-user experiments using the system, we had to rely more on the guidelines offered by the CDs. In this context, it is instructive to take a look at how the system holds up against a CDs-based evaluation. For each of the CDs, we first express it as a question [37] and then discuss it in the context of the FoXQ interface.

1. Abstraction gradient

What are the minimum and maximum levels of abstraction?

Can the fragments be encapsulated?

The system uses the document metaphor to shield the user, both from XQuery syntax (in the query-builder interface), and from XML data (in the result-viewer interface). It does not have user-defined functions or shortcuts.

2. Progressive evaluation

Can a partially-complete program be executed to obtain feedback on "How am I doing"?

While constructing an FoXQ query, users can click the "Run Query" button anytime to execute the partially-complete query. This feedback
enables the users to refine the query as required.

3. Consistency

*When some of the language has been learnt, how much of the rest can be inferred?*

FoXQ queries are constructed by the manipulation of document icons. This will be clear from the examples in Section 3.5. Once the users figure out how to create new icons on the screen, they can build *document rules* irrespective of the level of nesting of the XML data.

4. Hidden dependencies

*Is every dependency overtly indicated in both directions? Is the indication perpetual or only symbolic?*

Dependencies come into play only when the user is formulating a query that performs a join over multiple data sources. In such situations, we need mechanisms to distinguish between the different data sources, to specify the join conditions and to pick the result elements. In FoXQ notation, different data sources are denoted by different folder icons, displayed in different colors, in the selection pattern. When the user specifies a join condition on elements from different data sources, the elements take on the color of one of the sources (whichever one is picked first). Along similar lines, when the user specifies an element in the projection pattern, to avoid ambiguity, she can click on the element and the corresponding element in the selection pattern, and they both take on the same color. The second example in Section 3.5.3 illustrates this aspect in detail.
5. Premature commitment

Do programmers have to make decisions before they have the information they need?

One problem with the current implementation is that the users have to pick the URL (of the XML data) first, and then start building the query. There is no flexibility in the sequence of actions in this case. For example, the user cannot build a query first, and then try to see the results of execution against different XML data sources. In case the user made an error while entering the URL of the XML data, she would have to start all over again once the error is discovered. Another problem with the system is that we do not have any mechanism for inferring the DTDs in cases in which they are not available. The only way the user can figure out the structure of the XML data in such cases is by execution of a partially-complete query, and then inspecting the results and refining the query.

6. Error-proneness

Does the design of the notation induce ‘careless mistakes’?

The most common error we anticipate are the ones that arise from incorrect or incomplete understanding of the schema of the data source. This is not a problem with the FoXQ notation. Incidence of this error can be reduced by providing DTD-based help in building FoXQ queries.

7. Role Expressiveness
Can the reader see how each component of the program relates to the whole?

Once the users become familiar with the roles of the selection pattern and the projection pattern (see Section 3.1 for details), they should not have trouble understanding the part played by the different document icons in querying the data.

8. Viscosity

How much effort is required to perform a single change?

It is quite easy to make changes while the user is constructing the query, since she can choose arbitrary document icons and rename or delete them. Adding new icons to existing ones is also straightforward. The user can extend a simple pattern to a document rule by clicking on the “Rule” button. The changes a user can perform are limited to renaming or erasing document icons in the query-builder interface, and erasing, deleting or rearranging selected icons from the result data in the result-viewer interface. One limitation of the current implementation is that it does not allow the user to frame a query and then keep changing the URL to see results from different URLs. The interface “forces” the user to pick the URL of the data source, before the query can be built. This is a limitation of the current implementation of the interface, and not of the FoXQ language.

9. Visibility and juxtaposability

Is every part of the code simultaneously visible (assuming a large enough display), or is it at least possible to compare any two
parts side-by-side at will? If the code is dispersed, is it at least possible to know in what order to read it?

Yes. The rule is the entire code. The query-builder interface allows scrolling in case the user is building a query that cannot be displayed fully on a single screen. This situation would arise if the user were trying to formulate a join across many data sources, since queries expressing joins take up more screen real-estate. In addition to simple scrolling, the result-viewer interface also allows the user to drag the document icons around and arrange them any way they choose to. They can also delete unwanted data elements to “clean up” the data returned by query execution.

3.3 Constructing Queries in FoXQ

In this section, we first discuss the menu options in FoXQ and then describe how the user can construct FoXQ queries. Figure 3.2 shows the query-builder interface. The tools available to the user for constructing FoXQ queries are as follows.

1. **XML Data** When the user clicks on this button, a pop-up window comes up that allows the user to enter the URL of the XML data.

2. **Document** The user can select icons on the screen by clicking on them. The selected icons are highlighted by a red border. The *Document* button adds a document icon to every selected icon, except elements.

3. **Element** This button adds an element to every selected document icon, except folder icons.
4. Rule  This button allows the user to extend a simple selection pattern (one that has only the selection pattern) to a document rule (one that has both, a selection pattern, and a projection pattern).

5. Run Query  The user can click this button to execute the FoXQ query.

6. Erase  The Erase button allows the user to correct errors made during the construction of the FoXQ query. The user first selects the icons to be deleted, by clicking on them, and then clicks this button to erase them.

The user specifies the location of the XML data by clicking on the XML Data button and entering the URL. This brings up a single folder icon denoting the
root element of the XML data. The user can click on the top-left corner of the folder icon and enter the name of the root element of the XML data. Clicking on the folder icon highlights it, so that the user can then add document icons to it by clicking on the Document button. Document icons have a text area at the top where the user can enter the name that represents the tag of the corresponding XML element. As mentioned above, document icons can be added to other document icons or to folder icons. The user can also add elements to selected document icons. Elements have two text areas. The one on the left is for the name of the element, and the one on the right allows the user to specify a value. If only the name is specified, all the corresponding data is extracted from the XML data source. If both, the name and the value are specified, the element name is used to extract the data, subject to the restriction imposed by the value (this is done by means of a corresponding where condition in the generated XQuery).

The user can convert the simple query to a document rule by clicking on the Rule button. This generates another folder icon, which can be used to build the projection pattern. The projection pattern allows the user to control how the extracted data is displayed.

3.4 Viewing Results of FoXQ Query Execution

The FoXQ result viewer shown in Figure 3.3 allows the user to edit the result that is returned by the query. The user can reorder the data by moving the result “documents” around in the workspace. The system saves the resulting data in left–right, top–bottom order. The user can also delete selected elements in the result data by using the “eraser” tool. Another important feature is
that the user can generate new queries on saved result data. For example, a user could formulate a FoXQ query that extracts all the book information from multiple bibliography databases of his choice (using the join feature). The result generated by the execution of this query can be saved to a file on his local machine. The user can then formulate queries that extract more specific information (for example, titles of books written by a specific author) from the data saved on the computer. This query-on-query feature is helpful for two reasons.

- First, users can have local copies of data of interest to them. As a result, users do not need to connect to the web for subsequent data requests once they create a local repository. They would only need to connect to the web for periodic refreshes of their local repository.

- Second, the users can build further queries on the data stored on their
local machines. This feature gives the users some of the power of fully compositional semantics that is offered by XQuery since it allows them to build nested queries.

3.5 FoXQ Examples

To demonstrate the features offered by FoXQ we consider a few examples including some of the standard XQuery use cases [7]. The examples use a bibliography database with book and article (and other entries) each containing author and title information. The exact schema is not important here.

3.5.1 Selection Queries

These include queries that do a simple selection of data. The first example is a query that asks for the list of all the books (with all the details) available in the bibliography. This query can be expressed in XQuery as follows.

```
for $x1 in document("http://www.bn.com")/bib/book
return $x1
```

In the above query, the `document` keyword is used to specify the URL of the XML data, and the path expression `/bib/book` binds the variable `$x1` to the book nodes within the data. When `$x1` is specified in the `return` clause, it yields all the data at the `book` level. The same query can be expressed using FoXQ as shown in Figure 3.4. The query area shows a folder icon that is labeled with the root element of the XML document that is being queried. Inside of the folder icon, a document icon is displayed that represents elements to be retrieved from the data source. It is instructive to think about the document icon as a pattern that is matched against all the elements in the selected data source. In the
shown example, the use of the label "book" expresses the constraint that only book elements are to be retrieved.

![Book Element](image)

**FIGURE 3.4:** List of books in bibliography.

In order to extract only the author and title information from each book element in the XML data, the query shown in Figure 3.5 can be used. Note that in this case, the user specifies only the selection pattern, and the same pattern is used for projection. The corresponding XQuery query is as follows.

```xquery
<bib>
    return
      <book>
        { $x1/author }
        { $x1/title }
      </book>
  }
</bib>
```

Here, the variable $x1$ is bound to the book elements within the XML data. In the *return* clause, we use the binding for $x1$ to access the author, and title
elements, wrapped within the book tag. If required, the user can also specify renaming of tags using a projection pattern, as discussed in the next section.

3.5.2 Projection Queries

These queries are more powerful than selection queries because they consist of what is called a selection pattern for specifying the data to be selected, and a projection pattern for specifying how the data should be displayed.

To illustrate the pattern matching/selection process a little more we consider a slightly modified version of the query (Q1) from the use cases [7] where a user searches the bibliography data source for a list of titles of books published by Addison-Wesley in 1991.1 Assuming the user expects the list of book titles to be returned, she can express the query in XQuery as follows.

1 The original query in the use cases asks for the books published since 1991. In the current implementation, elements are always compared for equality. It is not difficult to extend FoxQ to allow the selection of other comparison operators, for example, via a pop-up menu. For simplicity and ease of use, we work with the modified query because using other operators impacts the usability.

```xml
<books>
  {
    for $x1 in document("http://www.bn.com")/bib/book
    where $x1/publisher = "Addison-Wesley" and $x1/year = "1991"
    return <result>
      { $x1/title }
    </result>
  }
</books>
```

In XQuery, variables are bound by for loops to nodes in the XML tree. The variables can then be used in the construction of the result tree and to express selection constraints. In the above query, the variable $x1 is bound to "book" nodes which are children of the "bib" node. The "book" nodes have "title", "year", "publisher", and "author" as their children. To express the selection conditions, the "publisher", and "year" nodes are accessed using the variable $x1 by the XPath expressions $x1/publisher and $x1/year. In the return
clause, the XPath expression $x1/title$ constructs the list of book titles.

The same result can be obtained by the equivalent FOXML query that is shown in Figure 3.6. The projection pattern specifies that the user wants only the book titles to be returned, each wrapped as a separate "result" document. By choosing the name "books" for the folder icon a new (local) XML file is created whose root node is "books". If the user did not want to express a projection, the selection pattern alone would have sufficed to express the query.

The semantics of document patterns and rules is defined so that a single pattern $P$ (that is, a selection without a separate projection pattern) is interpreted as a rule $P \Rightarrow P$, which means that the selection pattern is also used to determine the elements in the result. While this behavior is in many cases appropriate, in some cases it is not. Consider, for example, a query that tries to locate all books from, say 1991. A selection pattern containing a book element with only a year sub element and the query value 1991 would cause the resulting elements to just contain the year field.

![Diagram](image)

FIGURE 3.7: Books from a particular year.
Instead of requiring a separate projection pattern, FOxQ allows the user to mark fields by a little bar, so that they are interpreted only for generating a selection condition, but not as a projection field. For example, the query shown in Figure 3.7 is translated into the following XQuery code.

```xquery
where $xl/year = "1991"
return $xl
```

In the query shown in Figure 3.8, the user renames all the occurrences of “book” tag in the selected data to “result”. The user also renames the outermost tag from “bib” to “results”. This generates the following XQuery, similar to the example shown in Figure 3.5, except for the renamed tags.

```xquery
<results>
{
  return
    <result>
    { $xl/author }
    { $xl/title }
  </result>
}
</results>
```

### 3.5.3 Restructuring Queries

As a more advanced example, we consider a query that involves grouping of data. We demonstrate this feature by using the query (Q4) from the XML use cases. In this example, for each author in the bibliography, the query should return a list of the author’s name and the titles of all books by that author,
FIGURE 3.8: Renaming of "book" tag to "result".

grouped inside a "result" element. The query can be expressed in XQuery as follows.

```xml
<results>
{
  for $x1 in distinct-values(
  return
    <result>
      { $x1 }
      {
        for $x2 in
        where some $x3 in $x2/author
        satisfies deep-equal($x3,$x1)
        return $x2/title
      }
    </result>
}
</results>
```

In this case, $x1$ is bound to distinct values of the node "author". In the return clause of the outer query, we see that $x1$ is included since we want the titles grouped together under the author names. We also see the inner query where $x2$ is bound to the "book" nodes and then $x3$ is bound to "author" nodes.
under $x_2$. It is then checked whether $x_3$ matches $x_1$. The "deep equal" function matches trees. This ensures that only instances of $x_3$ and $x_1$ that match to full depth are considered. The "title" nodes under $x_2$ that meet the match criterion are then returned by the inner query.

The equivalent FoXQ query is shown in Figure 3.9. The "author" and "title" fields are exposed under "book" in the selection pattern to be referred to by the projection pattern, which nests a document icon for authors inside a "result" document icon. The fact that the "title" element is shown inside of the "author" element is interpreted as a grouping operation.

As another example, we look at a query that extracts a list of titles of all the books found both at amazon.com and bn.com. This demonstrates the join facilities of XQuery.
FIGURE 3.10: List of titles found both on amazon.com and bn.com.

```xml
<Books>
{
  for $x1 in document("http://www.bn.com/bib.xml")/book,
  $x2 in document("http://www.amazon.com/reviews.xml")/entry
  where $x1/title = $x2/title
  return
  <titles>
    { $x1/title }
  </titles>
}
</Books>
```

In FoXQ, in cases where only one data source has been specified, the document icons are displayed in black (this is the default color). When more than one data source has been specified, additional colors are used to differentiate between their document icons. Elements from different data sources participating in the same join condition would be displayed in the same color.
In this example, $x_1$ is bound to the book elements on bn.com and $x_2$ is bound to the review-entry elements on amazon.com. The “title” sub elements of both are compared and only those instances are returned for which the title occurs on both data sources. This can be achieved by the F0XQ query shown in Figure 3.10. In this case, the user specifies the join condition between the two data sources interactively, by selecting the corresponding elements. The elements from the two data sources participating in the same join condition would be displayed in the same color. The “title” element in the projection pattern would be ambiguous if we did not specify whether we need the titles from amazon.com or from bn.com. The user can remove this ambiguity by choosing the element in the projection pattern and then clicking on the corresponding element in the selection pattern. Both the elements would then be displayed in the same color as the element in the selection pattern, thereby showing which data source the data is being extracted from. This would also make it clear to the user which element from the selection pattern is being used in the projection pattern. Note that in this example, specifying the actual binding is not necessary since we are extracting titles that occur at both sites, but in more general cases, we would have to remove the ambiguity.

3.6 Future Work

As mentioned earlier, we do not aim to replicate all the functionality of XQuery in F0XQ. We are more interested in developing a more user-friendly, visual notation for the subset of XQuery that end users might commonly use. Some of the XQuery features we have excluded from F0XQ are discussed in this section. A few of these could easily be supported through extensions to F0XQ. Most
of the other, more complex, features of XQuery cannot be included in FoXQ without compromising on its usability. We also discuss in this section some other desirable features which might be incorporated into later versions of the system.

3.6.1 **Support for XML Attributes and Empty Elements**

FoXQ only supports simple XML elements. The elements can be arbitrarily nested and can also have textual data. The two other possible variations of elements, shown below are not supported.

1. Element with attribute.

   `<book year="1991">`

   `...`

   `</book>`

2. Empty element.

   `<img src="sample.jpg" />`

However, end users might find the idea of attributes confusing. Therefore, giving attributes a representation different from that for simple elements, might actually do more harm than good. On the other hand, discarding support for attributes altogether takes away the users option of querying them. This problem is, as of now, unresolved.
3.6.2 Support for XQuery Functions

The current version of the system does not support XQuery functions. The primary reason for excluding functions is that we have not done any end-user studies to choose a suitable visual representation for functions that is consistent with the document metaphor. The representation we are considering is the use of function cards to denote functions. Each supported function would have a corresponding function card that the user could pick from a palette, and place over the document icons representing data. This would result in the function being performed on the set of documents the function card overlaps with. Once a suitable visual notation has been picked, we must be careful to include only those functions that are commonly used by end users. Providing support for the entire range of functions available in XQuery could affect the usability of the system by cluttering the menus.

3.6.3 DTD-Based Help in the Query Builder

As mentioned at the beginning of this chapter, DTD-based help would enable the users to build syntactically correct queries in the sense that they are guaranteed to match the input data. We plan to extend this functionality to those cases in which the DTD for the data source is not available, by generating the DTD by using a tool like Xtract [29]. It might be useful to include a DTD browser (similar to the directory tree in Windows), which would enable the user to get a better overview of the DTD. Additional capabilities could be provided through the browser, whereby the user could double-click on any element and the corresponding FoXQ query could be automatically generated.
3.6.4 **Context-Sensitive Help in the Query Builder**

In the current version of the system, the user can execute the query at any stage, and evaluate the result to gauge progress. The usability of the system would be greatly increased if the user could seek context-sensitive help at intermediate stages of query construction. This could include some sample queries, based on the stage of the query being constructed, and the data source being queried.

3.6.5 **Support for Flattening Queries**

Query (Q2) in the XML use cases discusses the generation of author-title pairs from the XML data. For example, if a book has three authors, the query would generate three author-title pairs, one each, corresponding to each of the three authors. The following query performs flattening.

```xml
<results>
{ 
  for $b in 
    $t in $b/author,
    $a in $b/title
  return
    <result>
    { $t }
    { $a }
  </result>
} 
</results>
```

This query cannot be expressed in FoXQ since it does not allow arbitrary bindings. For the flattening to be possible, we need variables to be bound to the author and title elements independently of their parent. This is not
supported in the current version since we do not have a notation to distinguish between flattening and union.

3.6.6 Usability Testing

The strengths or weaknesses of any end-user programming language can only be really judged on the basis of feedback from the end users. As a next step, our efforts would be directed towards doing task-oriented experimentation with end user subjects. In the experiments, we could ask users to develop queries for specific searches. This would enable us to identify the shortcomings of the system. We could also perform experiments to compare how well users perform when the use FoXQ as opposed to XQuery. One group could develop queries using FoXQ and the other group would have to use a textual interface to XQuery. We could measure performance of the users on the basis of accuracy of the developed queries and the time taken to develop correct queries.
CHAPTER 4

TRANSLATION FROM FoXQ TO XQUERY

FoXQ patterns can be represented as trees as shown in Figures 4.1 and 4.2. The nodes of these trees might or might not be elements in the XML data being queried. Whenever the data source being queried provides a DTD, we can ensure through the user interface that all the nodes from the selection pattern are present in the XML document being queried. These nodes, which are associated with XML data, are called data nodes. On the other hand, the user has always the freedom to introduce additional nodes in the projection pattern for wrapping the data to be returned. These nodes do not have corresponding XML elements in the XML data being queried. We refer to them as grouping nodes. To access the data associated with an XML element in XQuery, we have to bind a variable to it. This is typically done in the for clause that binds a variable to all sub elements of a given element in an XML tree. Therefore, we represent a variable and the node it is bound to by a pair, which we refer to as variable/node binding in the following discussion.

In this section, we use the FoXQ queries shown in Figures 3.6 and 3.9 as examples to explain the translation from FoXQ to XQuery.
4.1 Inferring Regrouping Operations

As a first step in the translation process, we check if the user requests any regrouping of the data. Regrouping is inferred by comparing the selection and projection patterns to see if the relative depths of data nodes in selection and projection patterns have changed (we ignore the grouping nodes possibly injected by the user in the projection pattern). This can be determined by traversing the tree representation of the selection and projection patterns.

For the query in the Figure 3.6, whose tree representation is shown in Figure 4.1, we observe that the selection pattern has “bib” as the root node, with “book” as the only child node, which has “publisher”, “year”, and “author” as its children. The projection pattern has “books” as the root node. We ignore this node, as far as identification of the grouping is concerned, since it has been introduced by the user. We can assume that it is not a data node since it is not present in the selection pattern because we require all the data nodes to be specified in the selection pattern. The element “books” has “result” as its only child node, which in turn, has “title” as its only child node. The element “result” is not a data node either since it is not present in the selection pattern.
So we see that none of the parent-child relationships established in the selection pattern have been altered in the projection pattern. Therefore, we infer that this query does not require any regrouping of data.

For the query in Figure 3.9, whose tree representation is shown in Figure 4.2, we see that the selection pattern has “bib” as the root node, with “book” as the only child node, which has “author” and “title” as its child nodes. The projection pattern has “result” as the root node. This is not a data node since it is not present in the selection pattern. The “result” element has “author” as its child node, and “author” has “title” as its child node. In this case we see that there is a parent-child relationship between the “author” node and the “title” node in the projection pattern whereas they were at the same depth in the selection pattern. Hence we conclude that this query requires regrouping of the data. As explained in the example in Section 3.5, the user has used this query to obtain the titles grouped under the corresponding author name.

Once we have determined whether or not a query involves regrouping of the
data, we use the selection and projection patterns to generate the bindings to select the data as described in the next subsection. For queries that involve regrouping, we apply the method for regrouping described in Section 4.3.

4.2 Translation of Selection and Projection Queries

From the discussion of the XQuery examples in Section 3.5, we see that queries are evaluated by binding variables to the different XML nodes and then using the bound variables to access the data. We use a similar approach in translating FoXQ to XQuery. The translation uses the selection and projection patterns to generate variable/node bindings, which are to be used in the generated query for accessing the XML data.

Before the generation of the variable/node bindings, we traverse the projection pattern and identify the nodes for which the associated data has to be returned. It is obvious that these nodes or a common ancestor would require a variable to be bound to it. Nodes that are used to express conditions also require bindings to be used in the generated where clause. The translation algorithm generates the minimal set of bindings that meet these two conditions by computing least common ancestors in the tree representing the pattern.

As an example, we consider the selection pattern shown in Figure 4.1. Assume that the projection pattern requires that all the information associated with each book be returned, subject to the condition that the year of publication is “1991” and the publisher is “Addison-Wesley”. A naive approach to generating an XQuery query in this case would generate the for clause with all the bindings and then have only the binding for “book” node in the return clause.
In contrast, our translator first traverses the projection pattern and determines that the “book” node needs to be bound to a variable. It then traverses the selection pattern and ensures that the nodes in the `where` clause (“year” and “publisher” in this case) can be accessed by using the binding for the “book” node. This is possible since “book” is a parent of the “year” and “publisher” nodes in the selection pattern. As a result, the system generates the bindings necessary for accessing the data associated with the “book” node. The resulting, automatically generated query is as follows.

```xml
where $x1/year = "1991" and
    $x1/publisher = "Addison-Wesley"
return $x1
```

For any FoXQ query, once the variable/node bindings have been generated, the conditions for the `where` clause of the query are inferred from the nodes in the selection pattern that have data associated with them. In the case of the query in Figure 3.9, we see that there are no `where` clauses to be created. On the other hand, in the case of the query in Figure 3.6, we see that the data returned should have the publisher equal to “Addison-Wesley” and the year equal to “1991”. Therefore, the selection pattern from the query in Figure 3.6 also generates the following `where` conditions:

```xml
$x1/publisher = "Addison-Wesley"
$x1/year = "1991"
```

If a query does not involve regrouping of the data, we can directly build the `return` clause from the projection pattern. While traversing the projection pattern to build the `return` clause, for every data node we encounter, we return the corresponding bound variable, and for every grouping node, we return the
opening and closing XML tags with that node name. For example, for the query from Figure 3.6, the following return clause is generated.

```
<result>
  { $xl/title }
</result>
```

### 4.3 Translation of Restructuring Queries

In the projection pattern of the query in Figure 3.9, we see that “result” is a grouping node since it is not present in the selection pattern. On the other hand, “author” and “title” are data nodes. More importantly, “author” and “title” were at the same depth in the selection pattern. So we infer that we need to group the titles under the corresponding authors since “author” is located higher in the projection pattern than “title”. Since the binding for “author” cannot be used to access the data for “title” (since “title” is not a child of “author” in the XML data), we see that we need at least two separate bindings—one for “author” and another for “title”. Figure 4.2 shows the variable/node bindings for the query from Figure 3.9. Since XQuery does not have a groupby clause, grouping has to be implemented through nested queries. We use the XQuery keyword `distinct-values` to generate unique values of the element under which the grouping is being done (the “author” node in this case). Therefore, based on the selection pattern and the inferred regrouping, we generate the for clause of the outermost query as follows.

```
for $xl in distinct-values(
```

The outermost query does not have a where clause since no condition has been specified in the selection pattern. The return clause of the outermost query
ensures that the results get wrapped in a “result” tag. It also generates the author list using $x1 from above since it has been bound to the distinct values of bib/book/author which in turn is the list of authors (that possibly contains duplicates). So we obtain the following return clause for the outer query.

```
return
  <result>
    { $x1 }
    { ... }
  </result>
```

The three dots denote the nested query that has to be generated from the projection pattern. For the nested query, we have to build a new set of variable/node bindings to facilitate the computation of the join. This results in the following for clause:

```
```

The join condition is realized by a where clause for the inner query that compares $x2/author (the binding for “author” element in the inner query) to $x1 (the variable bound to the “author” element in the outer query). Since the join element is a structured element, the comparison has to be performed through the tree comparing operator deep-equal, which leads to the following where clause.

```
where some $x3 in $x2/author satisfies deep-equal($x3,$x1)
```

The translator consults the DTD of the queried data source to decide whether to create a tree comparison or a flat value comparison using “=”. If no DTD is available, the appropriate comparison operation is inferred from the visual query. For example, a condition like $x1/publisher = "Addison-Wesley" is
created because the query form defines (or better, requires) `publisher` to be an unstructured element. In cases when no information is available, the full tree comparison is employed, which is a safe strategy because it behaves like normal comparison for unstructured elements.

To complete the example query, the `return` clause needs to return the list of titles. This is achieved by returning `$x2/title`, which is the binding for the “title” nodes in the inner query. The complete generated query is shown below.

```xml
{ 
  for $x1 in distinct-values  
  return  
    <result>  
      { $x1 }  
      {  
        for $x2 in  
          where some  
            $x3 in $x2/author satisfies deep-equal($x3,$x1)  
          return $x2/title  
        }  
    </result>  
} 
```
CHAPTER 5
SYSTEM IMPLEMENTATION

5.1 System Architecture

The FoXQ interface has been developed in Java so that users can download it to their machines and run it if they have the Java runtime environment installed. The implementation details are discussed in Section 5.2. The FoXQ-to-XQuery translator has been implemented in Haskell. The Haskell functions that carry out the translation will be discussed in Section 5.3. The generated XQuery is processed by the IPSI-XQ engine [36]. This approach provides two different distribution architectures for the FoXQ software.

1. A user can run the Java applet from a browser, and the translator and the XQuery engine can run on a central server. This scenario is shown in

![FIGURE 5.1: FoXQ implementation architecture.](image-url)
2. A user can install all the binaries on her or his computer so that the processing is done locally.

The implementation choices were made keeping in mind the need for rapid prototype development. The user interface was designed with the primary objective to be easy to use. In spite of performance considerations, we have chosen Java because this approach enables an applet version of the interface which allows users to perform XML queries through their web browser. Haskell was chosen for implementing the translator from FoXQ to XQuery, because it allowed us the easy customization of the translator with regard to different query inference algorithms, join-generation strategies, etc. For execution of the generated query, we have chosen the IPSI-XQ engine that turned out to be simple, stable, and light weight. However, principally any XQuery engine that can be run on the webserver (for the first scenario) or on the user’s computer (for the second scenario) would be suitable for this task.

5.2 The FoXQ Interface

The FoXQ interface has been developed in Java, keeping portability in mind. Currently, we have implemented only the application version and not the applet version. The system can be downloaded by the user and run from the user’s machine. The FoXQ queries will be executed on a central server. In this case, the translator module and the XQuery engine reside on the College of Engineering web server. The FoXQ system can be divided into the two main components discussed below.
5.2.1 The Query-Builder Component

This component allows the user to build FoXQ queries. Typically, the user would select the URL of the data source first and then build the query using the menu options provided on the panel on the left. The main classes in the FoXQ query builder component are discussed below:

1. FoXQFrame class This is the outermost frame that has all the elements necessary for creation and editing of FoXQ queries. On the panel on the left, it has the options for building the queries. On the panel at the bottom, it has the file-related options. The panel on the right is meant for a DTD browser which would allow the user to view the DTD while building the queries. This feature has not been implemented in the current version.

2. ElementComponent class Instances of this class are used to denote the leaf elements in the data. This has a text area for the name of the leaf node and another text area for the textual data (corresponding to the PCDATA elements of XML data).

3. CardComponent class Instances of this class are used to represent the different levels of nesting in the data being queried. Typically, a CardComponent object is used to represent an inner node in the tree representation of the XML data.

4. FolderComponent class This is the component that acts as a container for CardComponent elements. In case of simple selection queries, we will have a single instance of this class and in cases where we have selection-
projection queries, we will have two instances of this class (one for the selection pattern and the second one for the projection pattern).

5. RuleComponent class In cases where the user is developing a selection-projection query, this component acts as a container to hold the two FolderComponent instances required.

5.2.2 The Result-Display Component

This component displays the results returned by the query execution. We re-use the document metaphor for this purpose and display the results using components similar to the ones used by the user for building the query. We are aware that the user might not require all the data that is returned by the query. The user might also want to modify the order of elements in the data. We have provided additional functionality in the display component that allows the user to move the components around on the screen so that he can rearrange the elements. The user also has the option of deleting components in the result data. After all the necessary changes have been made, the user can save the edited result. The result-viewer interface has the following classes.

1. DisplayFrame class This is the frame that displays the results returned by the query execution. The aim of this interface is to shield the end user from the textual or tree representation of XML data.

2. DElementComponent class Objects of this class look the same as objects of the ElementComponent class. The main difference is that instances of this class are used by the DisplayFrame component only, whereas instances of the ElementComponent class are used only by the FoXQFrame
component. The instances of this class have the added functionality that they can be moved (by the user by dragging with the mouse) within the enclosing DCardComponent container.

3. DCardComponent class  Along the lines of the differences between the DElementComponent class and the ElementComponent class, the DCardComponent class is used only by the DisplayFrame component whereas the CardComponent class is used by the FoXQFrame component. The instances of this class can be rearranged within the enclosing DFolderComponent component by dragging with the mouse.

4. DFolderComponent class  This is the class that acts as the container for the DCardComponent instances. This also represents the root element of the XML data.

5.3 The Translation Module

XML data can be represented as trees. In the translation module (implemented in Haskell), FoXQ queries are internally represented as trees, using the Tree datatype.

```
data Tree = Node Tag [Tree]  
           | Leaf String
```

Unless otherwise mentioned, we use p for the selection pattern and q for the projection pattern. Individual nodes in the selection and projection patterns are represented by elements of the Tag type which is a type synonym for String. The nodes in p and q do not have to be unique. That is, two nodes can have the same name in the selection pattern or the projection pattern or both.
uniquely identify a node, we represent each node by a list that contains all the
tags (in sequence) from the root of the tree to the node under consideration.
We introduce a type \texttt{Node} for nodes and a function \texttt{nodes} that yields the list of
all nodes contained in a tree.

\begin{verbatim}
  type Tag = String
  type Node = [Tag]
  type Var = String

  noNode :: Node
  noNode = []

  nodes :: Tree -> [Node]
  nodes (Leaf _) = noNode
  nodes (Node a ts) = [a]:map (a:) (concatMap nodes ts)
\end{verbatim}

This list representation also allows us to identify the common ancestor of a set
of nodes. Along similar lines, we use the function \texttt{leaves} to extract all the leaf
nodes from a pattern and the function \texttt{leaf} to extract all the leaf tags from a
pattern.

\begin{verbatim}
  leaves :: Tree -> [Node]
  leaves t = filter (\n -> length (pre n) == 1) ns
    where ns = nodes t
          pre n = filter (\m -> m 'intersect' n == n) ns

  leaf :: Tree -> [Tag]
  leaf (Node a [Leaf _]) = [a]
  leaf (Node a []) = [a]
  leaf (Node a ts) = concatMap leaf ts
\end{verbatim}

We use the type \texttt{Expr} to represent XQuery expressions. Syntactically cor-
rect XQuery statements are generated through a specialized \texttt{Show} function that
transforms the datatypes Flwr and Expr into string. We illustrate this by showing some (simplified) XQuery grammar, the corresponding Haskell type, and the Show instance.

\[
\begin{align*}
\text{Expr} & : = \text{FLWRExpr} \\
& \quad | \text{AndExpr} \\
& \quad | \text{PathExpr} \\
& \quad | \text{GeneralComp} \\
\text{FLWRExpr} & : = \text{ForClause}^+ \text{ WhereClause}? \text{ Return Expr} \\
\text{AndExpr} & : = \text{Expr} \text{ And Expr} \\
\text{PathExpr} & : = \text{StepExpr}(\text{Slash StepExpr})^* \\
\text{GeneralComp} & : = \text{Expr} \text{ Equals Expr}
\end{align*}
\]

Based on the above grammar, we can have a datatype for Expr, with the corresponding type constructors as follows.

\[
\begin{align*}
\text{data Expr} & = \text{Flwr Flwr} \\
& \quad | \text{And Expr Expr} \\
& \quad | \text{PExpr Parent Child} \\
& \quad | \text{Comp Expr Expr}
\end{align*}
\]

In the simplified form, Parent and Child are strings that hold the names of the parent and child nodes that are involved in the path expression. In the translator module, we define the type of arbitrary FLWR expressions as follows.

\[
\begin{align*}
\text{data Flwr} & = F [(\text{Var,Expr})] [\text{Expr}] [\text{Expr}]
\end{align*}
\]

\(F\) is the constructor, followed by a list of tuples denoting the variables and the path expressions they are bound to in the for clause. The second list is composed of expressions that participate in the where clause. The third list has the expressions for the return clause. The query output is controlled by the Show instance given below.
instance Show Expr where
    show (Flwr f) = show f
    show (And e1 e2) = show e1 ++ " and " ++ show e2
    show (PExpr p c) = p ++ "/" ++ c
    show (Comp e1 e2) = show e1 ++ " = " ++ show e2

instance Show Flwr where
    show (F fs ws rs) | length ws > 0 = showFL fs ++ "\n where " ++ showWL ws ++ "\n return " ++ showRL rs
                      | True = showFL fs ++ "\n return " ++ showRL rs

showFL, showWL and showRL are functions that print out the for, where and return clauses respectively.

5.3.1 Inferring Regrouping Operations

To infer the regrouping operations, we need to traverse the selection and projection patterns and verify if tags at the same depth in the selection pattern have been moved to different depths in the projection pattern. In order to do this, we traverse \( p \) first and assign tags the depth they have in a given tree. This results in all the tags at the same depth being assigned the same number. We start the depths at 0. This is done by the function \( \text{depths} \) shown below.

\[
\text{type Depth = Int}
\]

\[
\text{depths :: Depth -> Tree -> [(Depth, Tag)]}
\]

\[
\text{depths _ (Leaf _)} = []
\]

\[
\text{depths d (Node a ts) = (d, a) : concatMap (depths (succ d)) ts}
\]

Given an integer to start the count from and a tree representation of a pattern, \( \text{depths} \) gives us a list of pairs. In each pair, the first value is the depth (from the root node) and the second value is the name of the node. We run \( \text{depths} \) twice,
once for the selection pattern and the second time for the projection pattern and compare the lists generated. The nodes in the selection pattern that are at the same depth must be at the same depth in the projection pattern too. If this is not the case, we conclude that the data has to be regrouped.

5.3.2 Translation of Selection and Projection Queries

To identify the data nodes, we need to build a list of nodes in the selection pattern. This is done by the function \( \text{tags} \).

\[
\text{tags :: Tree -> [Tag]}
\]

\[
\text{tags = map last . nodes}
\]

As discussed at the beginning of Chapter 4, we assume that all the nodes in the selection pattern have corresponding nodes in the XML data. The projection pattern includes the data nodes whose data has to be returned. We determine the common ancestor for the nodes included in the where and return clauses, so that we can bind a variable to the ancestor node and use that binding to access the data. To identify the nodes participating in the where clause, we use the function \( \text{wnodes} \) to step through the selection pattern and mark the Leaf nodes. The inner nodes are wrapped in the NTag constructor and the leaf nodes are wrapped in the LTag constructor. The where clause is inferred from the nodes of the selection pattern that contain text (represented by a Leaf node). The function \( \text{whereNodes} \) can then extract the marked nodes, and \( \text{buildW} \) builds the where clause.

\[
\text{data QTag = NTag Tag}
\]

\[
| \text{LTag Tag}
\]

\[
\text{deriving Show}
\]
wnodes :: Tree -> [[QTag]]
wnodes (Leaf s) = [[LTag s]]
wnodes (Node a ts) = [NTag a]:map ((NTag a):)
                     (concatMap wnodes ts)

whereNodes :: Tree -> [Node]
whereNodes t =
    map unwrap (filter (\x -> isLeaf (last x)) (wnodes t))

unwrap :: [QTag] -> Node
unwrap [] = []
unwrap ((LTag x):xs) = unwrap xs
unwrap ((NTag x):xs) = x:unwrap xs

isLeaf :: QTag -> Bool
isLeaf (LTag _) = True
isLeaf (_) = False

buildW :: [[QTag]] -> Node -> [Expr]
buildW [] _ = []
buildW (q:qs) n =
    EComp (PExpr "$x2" (path (unwrap q \ n)))
    (DExpr cond):buildW qs n
    where (LTag cond) = last q

The function parent accepts the selection and projection patterns as inputs
and returns the path to the common ancestor in the selection pattern of all
those leaves (in both patterns) that represent data nodes and the leaves that
represent where conditions.

parent :: Tree -> Tree -> Node
parent p q = foldr1 intersect ns
    where ws = whereNodes p
          rs = filter (\x -> last x 'elem' leaf q)
                 (leaves p)
          ns = union rs ws
The function path builds an XPath expression for a node.

```haskell
path :: Node -> String
path [] = ""
path (a:as) | length as > 0 = a ++ "/" ++ (path as)
            | otherwise = a
```

Given the selection pattern, we build the return clause and the where clause independent of each other. Once we determine the common ancestor using parent, we generate a variable/node binding for the ancestor node in the for clause and use the binding in the return and where clauses. In building the return clause, we need to keep track of the data nodes and the grouping nodes. When we encounter grouping nodes in the projection pattern, we need to inject the corresponding tags in the generated XML data. On the other hand, when we encounter data nodes as leaves, we need to use the variable/node binding to access the data associated with that node.

The function return accepts the projection pattern and the node bindings as inputs and generates a list of expressions that compose the return clause. The function isin categorizes the nodes as data nodes (if they are bound) or as grouping nodes (if they do not have a binding). The function buildBind generates the bindings, given the list of nodes and the parent node (output from parent).

```haskell
return :: Tree -> [(Tag, Expr)] -> [Expr]
return (Node a []) ys = [isin a ys]
return (Node a [Leaf _]) ys = [isin a ys]
return (Node a ts) ys = RTag a:
  concatMap (flip return ys) ts

isin :: Tag -> [(Tag, Expr)] -> Expr
isin a ys = maybe (RTag a) id (lookup a ys)
```
buildBind :: [Node] -> Node -> [(Tag,Expr)]
buildBind [] _ = []
buildBind (n:ns) m = (last n,PExpr "$x2" (path (n \ m))):(buildBind ns m)

The FoXQ interface passes the internal representation (as the Haskell datatype Tree) of the selection and projection patterns, and the URL of the XML data source as parameters to the translation module. We always bind the variable $x_1$ to the URL using the document keyword. We also generate a second binding using the variable $x_2$ to the node identified as the common ancestor of the nodes in the where and return clauses. Putting it all together, we have the following function transSel for the translation of selection and projection queries.

transSel :: Tree -> Tree -> Flwr
transSel p q = F [("$x1",DExpr ("document("++url++")")),
    ("$x2", (PExpr "$x1" pp))] ws rs
where par = findParent p q
    pp = path par
    ws = buildW (filter (\x -> isLeaf (last x))
                 (wnodes p)) par
    rs = return q (buildBind (leaves p) par)

5.3.3 Translation of Restructuring Queries

Regrouping is achieved by nested queries. To generate the nested query, we have to identify the nodes under which grouping is being done, generate the corresponding data using the outer query, and generate the grouped data using the inner query. The function tagsum accepts the output from the function depths and generates a list containing lists of nodes at every depth/level in a given tree. The function tagsum, shown below, builds this list.
tagsum :: [(Depth,Tag)] -> [[Tag]]
tagsum xs = map (\x -> map snd x) ys
    where ys = groupBy (\x y -> fst x == fst y) xs

The function \texttt{extract} takes the output from \texttt{tagsum} and the result of application of \texttt{depths} on the projection pattern and returns the list of nodes involved in the regrouping. The auxiliary function \texttt{getTag} takes a node as input and returns the tuples showing all the depths at which the node occurs. The function \texttt{check} verifies if all the nodes in a list of \((\text{Depth}, \text{Tag})\) tuples are at the same depth.

\begin{verbatim}
extract :: [[Tag]] -> [(Depth,Tag)] -> [Tag]
extract [] xs = []
extract xs [] = []
extract (x:xs) ys | check zs = extract xs ys
    | otherwise = map snd zs ++ extract xs ys
    where zs = concatMap (\flip getTag ys) x

getTag :: Tag -> [(Depth,Tag)] -> [(Depth, Tag)]
getTag x ys = filter (\z -> snd z == x) ys

check :: [(Depth, Tag)] -> Bool
check xs = length (nub (map fst xs)) < 2
\end{verbatim}

The function \texttt{regrp} builds the regrouping queries and is defined as follows.

\begin{verbatim}
regrp :: Tree -> Tree -> Flwr
regrp p q = rebuild tlist p q glist
    where glist = extract nlist (depths 0 q)
        nlist = tagsum (depths 0 p)
        bound = tags p
        tlist = nodes r
\end{verbatim}

The function \texttt{rebuild} builds the nested query. The input to \texttt{rebuild} consists of
• The list of paths to the leaves in the projection pattern (generated by application of nodes to the projection pattern). This is actually a list of lists – one list per leaf node.

• The selection pattern.

• The projection pattern.

• The list of nodes involved in regrouping (generated by extract).

The output from rebuild is the translated XQuery.

```plaintext
rebuild :: [Node] -> Tree -> Tree -> [Tag] -> Flwr
rebuild [] p q gs = F [] [] []
rebuild [x] p q gs = traverse x p q gs
rebuild (x:xs) p q gs =
  F [] [] [Union (Flwr (traverse x p q gs))
            (Flwr (rebuild xs p q gs))]
```

In case the list of nodes has only one item, we simply apply traverse to the node. If the node under consideration has many children (more than one item), we apply traverse to the first child, and apply rebuild to the remainder of the list of children. All the result sets are then combined using the XQuery keyword union. The function traverse builds the outermost level for grouping and then makes a call to the function inTrav to build the inner levels of the nested query. The input to traverse is the same as that for rebuild except for the first parameter. In the case of traverse, it is a list that denotes the path to a particular leaf node. The variable url holds the URL of the XML data. This value is passed to the translation module from the FoXQ interface.
The function \texttt{inTrav} is defined similar to \texttt{traverse} and it builds the inner nested queries. The fifth parameter used in the call to \texttt{inTrav} is the list that stores the variable/node binding built so far – this list grows as we recursively apply \texttt{inTrav}, and is used in inner queries. We have a list of unbound variables stored in \texttt{varsx} and this is passed through successive applications of \texttt{inTrav}.

\texttt{inTrav :: Node -> Tree -> Tree -> [Tag] -> [(Var,Tag)] -> [String] -> Flwr}

\texttt{inTrav [] _ _ _ _ = F [] [] []}
\texttt{inTrav (x:xs) p q gs vs fs =}
\texttt{F [(v1,RExpr url ppar)] [NComp v2 (PExpr v1 s) o]}
\texttt{[PExpr v1 x, Flwr (inTrav xs p q gs ((v1,x):vs) vrem)]}
where ppar = path (findParent p q)
\begin{align*}
v1 &= \text{head } fs \\
v2 &= \text{head } (\text{tail } fs) \\
vrem &= \text{tail } (\text{tail } fs) \\
o &= \text{fst } (\text{head } vs) \\
s &= \text{snd } (\text{head } vs)
\end{align*}

5.4 File Format for FoXQ Queries and Results

All the objects the user manipulates in the FoXQ interface are instances of the classes discussed above. We have an internal XML representation of all the classes discussed above. We use this internal representation when the data or
FoxQ queries need to be written to files. This representation allows us to easily parse the document, and load the corresponding FoxQ query when the user wants to resume working on a saved query file. Table 5.1 shows a FoxQ query and the corresponding internal representation. The user also has the option of saving the data generated as a result of query execution, in XML format. Table 5.2 shows the data returned by query execution, and the corresponding internal representation. Loading the data from the file into the result viewer later causes data being parsed and displayed using the document metaphor. This ensures that the users never have to view XML data in text format (they could, if they so wish, directly open the saved files using a text editor and view and edit the data as plain text). The document metaphor is used consistently to shield the user from XML and XQuery syntax.
<table>
<thead>
<tr>
<th>FoXQ representation</th>
<th>XML representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>![FoXQ diagram]</td>
<td><code>&lt;ruleComponent&gt;</code></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
TABLE 5.2: Format for saving FoXQ results to file.

<table>
<thead>
<tr>
<th>FoXQ representation</th>
<th>XML representation</th>
</tr>
</thead>
</table>
| ![FoXQ representation](image) | <dFolderComponent name="results">
  <dCardComponent name="result">
    ...
  </dCardComponent>
  <dCardComponent name="result">
    ...
  </dCardComponent>
</dFolderComponent> |
CHAPTER 6

CONCLUSIONS

In this thesis, we have presented our work on a visual query language for XML. FoXQ has been designed keeping end users in mind. We have also implemented a FoXQ prototype that allows the user to develop FoXQ queries which then get translated into XQuery and executed by a query engine. The FoXQ interface then displays the returned results.

We believe the main advantage of FoXQ is that it is straightforward and easy to use. It supports the basic querying facilities a user at a beginner to intermediate level would require. The existing system can be extended to incorporate some of the more advanced features of XQuery. Since FoXQ is strictly an end-user language, full support for all features of XQuery is not too serious a consideration.

We realize that over-crowded menu-bars or cluttered option palettes can only serve to confuse end users. So we have kept the implementation of the language very simple and straightforward. We could extend the language to support XQuery functions but the implementation would have to be totally menu-driven with extensive help dialog. We should not expect the end-user to remember all the functions at his disposal.

The FoXQ notation is not limited or tied to any particular query language. We translate the FoXQ queries to XQuery simply because XQuery is the W3C standard. We could, as easily, translate FoXQ queries to any other XML query
language. Another reason for adopting XQuery is the growing availability of query processing engines. This choice has allowed us to concentrate on the language and interface design issues, and not worry about the implementation of a query processor.

XQuery is very powerful as a query language. However, to be used effectively, it requires a significant initial learning process. At an enterprise level, the designers/developers of XML applications might be experts at programming or databases. They would probably not have much trouble learning or using XQuery. End users do not have the time and sometimes do not want to learn a query language.

FoXQ's interface allows end users to progressively build or refine their queries without having to deal with text-based programming issues. It supports basic queries that would be developed by a typical end user. More complicated features of XQuery (for example, functions) would require a high level of sophistication from the end user.
**BIBLIOGRAPHY**


