XGI: A Graphical Interface for XQuery Creation

Xiang Li[1], John H. Gennari[1], PhD, and James F. Brinkley[1,2,3], MD, PhD

Structural Informatics Group
Departments of [1]Medical Education and Biomedical Informatics [2], Biological Structure, and [3]Computer Science and Engineering, University of Washington, Seattle, WA

Abstract

XML has become the default standard for data exchange among heterogeneous data sources, and in January 2007 XQuery (XML Query language) was recommended by the World Wide Web Consortium as the query language for XML. However, XQuery is a complex language that is difficult for non-programmers to learn. We have therefore developed XGI (XQuery Graphical Interface), a visual interface for graphically generating XQuery. In this paper we demonstrate the functionality of XGI through its application to a biomedical XML dataset. We describe the system architecture and the features of XGI in relation to several existing querying systems, we demonstrate the system's usability through a sample query construction, and we discuss a preliminary evaluation of XGI. Finally, we describe some limitations of the system, and our plans for future improvements.

Introduction

XML (Extensible Markup Language) has become the standard for the exchange and sharing of information among heterogeneous data sources. In January 2007 the W3C (World Wide Web Consortium) recommended XQuery (XML Query Language) [1] as the primary language for querying semi structured XML datasets because it is powerful and well-supported by the software development community. However, the inherent complexity of XQuery makes it intimidating for inexperienced biomedical researchers to effectively query different XML data sources. There is therefore a growing need for tools, such as graphical query interfaces, that can help inexperienced users create simple and accurate queries. In this paper we describe one such tool, called XGI (XQuery Graphical Interface).

An example

We motivate the need for XGI through an example XML database we developed as part of the UW Integrated Brain Project [2], a snippet of which is shown in Figure 1. This database consists of multiple neurosurgical patients whose areas of language cortex have been mapped through a surgical planning procedure called cortical stimulation mapping (CSM). In this procedure common objects are shown to awake patients in whom a portion of the skull has been removed. While the objects are being shown, various cortical areas are electrically stimulated while the patient is asked to name the objects. The locations of these cortical areas are marked by labeled tags called stimsites in Figure 1. If the patient makes an error while the site is stimulated then the type of error is recorded. For one particular stimulation trial in Figure 1 (trial 19) the patient made a semantic naming error, calling a squirrel a mouse, when site 31 was stimulated (as controls, some trials do not involve stimulation). Following surgery this error is coded as type 2, semantic paraphasia, by our collaborating researchers, who enter all this information in our database [3].

```
<root>
 <patient>
  <sex>M</sex>
  <pnum>50</pnum>
  <viq>85</viq>
  <age_at_registration>39</age_at_registration>
  <surgery>
   <csmstudy>
    <function>Object Naming</function>
    <trial>
      <trial_num>19</trial_num>
      <stimulated>Y</stimulated>
     <item>squirrel</item>
     <patient_response "mouse..."/>
     <stimsite>
       <site_label>31</site_label>
     </stimsite>
     <trialcode>
        <term>
          <fullname> semantic paraphasia />
          <abbrev>2</abbrev>
        </term>
     </trialcode>
    </trial>
   </csmstudy>
  </surgery>
 </patient>
</root>
```

Figure 1. XML database snippet

An example XQuery of this database (which currently has over 80 patients) is shown in Figure 2. This query asks for a list all those patients that made at least one semantic naming error for a stimulated trial.

```
<result>
{
  for $p0 in $root/patient
  where $p0/surgery/csmstudy/trial/trialcode/term/abbrev='2'
and
          $p0/surgery/csmstudy/trial/stimulated='Y'
  return
    <patient id='{$p0/pnum/text()}'>
      {$p0/sex}
     <age>{$p0/age_at_registration/text()}</age>
      <verbal_iq>{$p0/viq/text()}</verbal_iq>
    </patient>
}
</result>
```

Figure 2 Sample XQuery

For each of these patients the construct portion of the query (following the return token) creates an id attribute from the pnum element, and includes age and verbal IQ child elements, each renamed for readability (or possibly to conform to an external ontology).

```
<patient id="50">
  <sex>M</sex>
  <age>39</age>
  <verbal_iq>85</verbal_iq>
</patient>
<patient id="52">
  <sex>F</sex>
  <age>46</age>
  <verbal_iq>66</verbal_iq>
</patient>
```

Figure 3. Query results

A snippet of the results of running this query on our XML query system, called XBrain [4], is shown in Figure 3. Our goal is to develop graphical methods for generating the XQuery shown in Figure 2, so that end-users do not need to learn the complexities of XQuery.

Approaches to graphical XQuery generation

Query By Example (QBE) was the first graphical query language that enabled relational database users to query and modify data sources without having to learn all the complexities of the underlying query language [5]. Although much more work has been done for relational databases than XML, several tools have applied the QBE approach to assist users in formulating queries for XML data sources. These tools can generally be categorized by whether they use a structured or unstructured query approach. Elsewhere, we provide a more comprehensive review of these tools, as well as a more complete description of XGI [6].

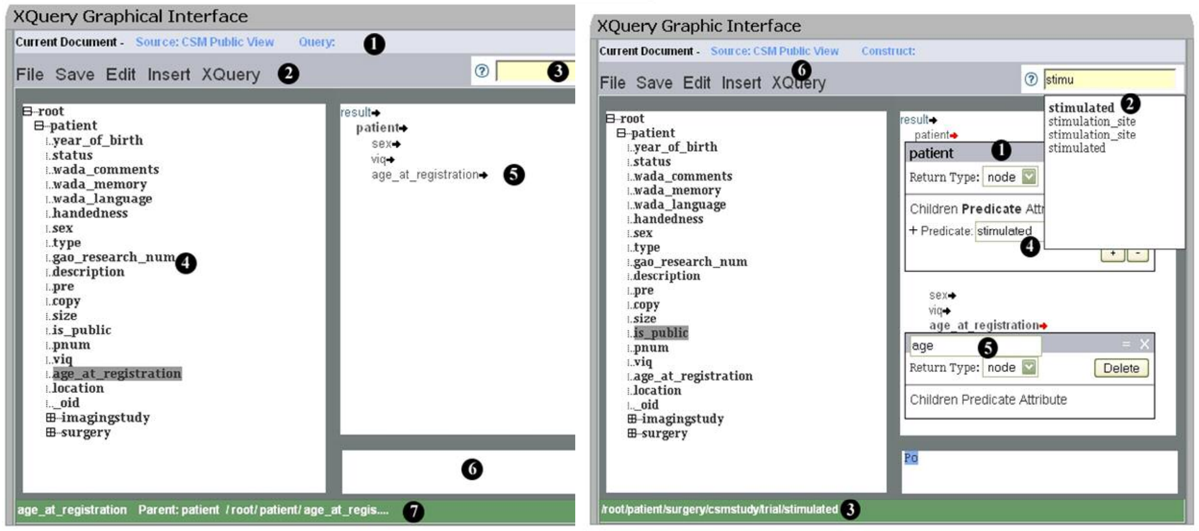
The structured query approach is characterized by the lack of arbitrary, hierarchical structures in the XML-querying results. Such an approach is exemplified by QURSED (Querying and Report Semi-structured Data), which is a query forms and reports (QFRs) generator [7]. In QURSED the graphical query interface is divided into two parts: the QURSED editor and the QFRs. The QURSED editor displays the source tree objects for developers to choose elements from the tree to create QFRs. The QFRs are form-based web front-ends of the query interface. The form elements are instantiated by end-users to create query set specifications (QSS), which are then compiled by the QURSED compiler and processed by the QURSED run-time engine to query XML data.

In the unstructured query approach users are allowed to generate arbitrarily formulated queries. This approach is exemplified by XQBE (XQuery By Example) [8], which consists of two components: the XQBE client and the XQBE server. The server translates the query result tree to the XQuery statement, executes the query over any arbitrary XML data source, and returns the result to the XQBE client. The client is a stand-alone, Java-based graphical query editor that allows users to construct the query result tree from any arbitrary XML source schema by explicitly defining both the source tree and the query result tree.

In general the structured approach is easier to use but less flexible, whereas the unstructured approach has the opposite tradeoff. XGI takes a middle ground by incorporating the best aspects of both approaches. XGI uses a web-based architecture that can reduce the cost of implementing a graphical query system by removing biomedical researchers from having to install and maintain software. XGI also provides users with a navigable source tree that assists users in understanding the source schema and gives users the ability to graphically choose elements from the source schema to be included in the query schema. XGI also has a robust query creation process that can help users to create expressive XQuery statements.

XGI system architecture

The XGI system is a web-based application that implements the Asynchronous JavaScript and XML (AJAX) framework to look and behave similarly to a desktop application. The system architecture includes a graphical query interface, a schema file manager, a data model controller, and an XQuery generation engine. The graphical query interface is a collection of JavaScript libraries designed to be used through a web browser in order to graphically inspect the source data schema, formulate the query schema, and create the XQuery statement. The schema file manager allows users to both load and save the source and the query schema file. The data model controller provides functions for the XGI system to manage users' access to the source and the query schema models. The XQuery generation engine uses an Extended Backus-Naur Form (EBNF) grammar to translate the query schema through a series of nested statements to its corresponding XQuery format.



a b

Figure 4. Screen shots of the XGI system: **a)** A portion of the CSM source schema and the XGI view of the query of Figure 2. **b)** Some of the pop-up boxes used during construction of this query.

XGI system use

In this section, we use XGI to graphically generate the example query shown in Figure 2.

In Figure 4a, we show the interface of the XGI system marked to indicate six major areas: the toolbars (1 & 2), the search box (3), the source panel (4), the constructed query panel (5), the saved predicate panel (6), and the information panel (7).

To construct the query we first select elements from the source tree in Figure 4a-4, which is a representation of the complete schema of the XML database snippet shown in Figure 1, with only top-level elements displayed. We add these elements, possibly renaming them, to the constructed query tree in Figure 4a-5, which corresponds to the return portion of the query in Figure 2. To only retrieve patient data when the patient has made a semantic naming error for a stimulated trial we need to construct two predicates and add them to the constructed patient element through the node's tooltip menu(Figure 4b-1).

To accomplish these tasks we first load the saved CSM source schema, called "CSM Public View" (from a File->Open Schema->Source menu item), which populates the source panel with a collapsible source tree (See Figure 4a-4). The full CSM schema has 473 elements in it, so the collapsible tree makes it easier to browse the source schema. Then we need to define a root for the generated query by adding a user-defined root node to the query tree, either through the "File->Insert->Root" dialog box, or by double clicking on any selected source tree node. Once a source tree node is added to the query tree, it forms an implicit mapping edge between the source and query tree node. Users can also add a user-defined node anywhere on the query tree to arbitrarily structure the query result. The user-defined node does not contain any mapping edge to a node in the source tree.

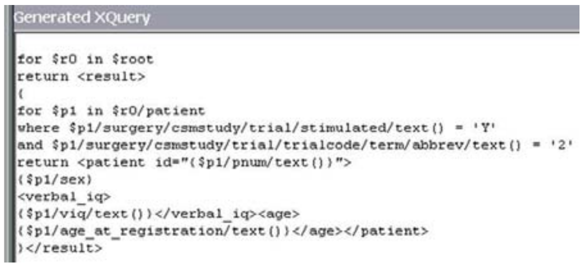
We add the source tree node "patient" to the query tree root. Then, we select the "patient" node and add "sex", "viq", and "age_at_registration" nodes from the source tree. If we want to add a user-defined node to "patient", we can add the user-defined node through the "Children" submenu in the tooltip menu (Figure 4b-1), or through the "Insert" menu.

Next we add the pnum of the patient as an attribute with the name "id" by opening the "Attribute" submenu first, then entering the name of the attribute ("id") into the name box and then adding the source tree node ("pnum") into value box. We also need to open the "Predicate" submenu and then place two predicates on it: 1) test whether the patient has a stimulated trial, and 2) test whether the patient made a semantic error on any trial.

To add the first predicate, in which we need to find out whether the patient had a stimulated trial, we utilize the unique search function of XGI to find the "stimulated" source element. Searching can eliminate the tedious and time-consuming task of finding the desired node in a large source schema tree. Nodes whose name begins with the search string are automatically returned (Figure 4b-2). To distinguish result nodes with the same name, XGI will display each result node's path information in the information panel when users position the mouse over the node (Figure 4b-3). Then we add the "stimulated" source tree node and the text comparison "Y" to the predicate field to add the predicate for "patient" (Figure 4b-4). For the second predicate, we use the search function again to find the appropriate source schema element that we need to test for if the patient made a semantic error during any trial. We select the "abbrev" source tree node and the text comparison "2" to the predicate field to add the second predicate.

Finally, we change the name of the node "age_at_registration" to just "age" and the name of the node "viq" to "verbal_iq". Users can change the name of the query node so the same query result will be returned under a different tag name. To rename the node, we open the node's tooltip menu and select the node's name on the top left corner, which will be replaced by a text field to enter the new name. Once we have changed "age_at_registration" to "age", (Figure 4b-5) the generated query node is changed to "age" automatically, with a similar result for "viq".

From the constructed query schema we generate the XQuery statement by clicking the "XQuery" button (Figure 4b-6), creating the result in Figure 5. Comparison of this generated query with the manually created query in Figure 2 shows that they are the same except for formatting and internal variable names, and in fact both return the results shown in Figure 3.



```
for $r0 in $root
return <result>
(
for $p1 in $r0/patient
where $p1/surgery/csmstudy/trial/stimulated/text() = 'Y'
and $p1/surgery/csmstudy/trial/trialcode/term/abbrev/text() = '2'
return <patient id="($p1/pnum/text())">
($p1/sex)
<verbal_iq>
($p1/viq/text())</verbal_iq><age>
($p1/age_at_registration/text())</age></patient>
)</result>
```

Figure 5. Generated XQuery

Evaluation and validation

To test our implementation, we used a library of saved queries from the XBrain project [4]. These 62 saved queries were used to access portions of the CSM database. They were "saved" because users deemed them either important or common enough to warrant reuse. This library of queries was created by users before XGI was designed or built.

As a preliminary evaluation of the usefulness of XGI, we assessed how many of these saved XQuery statements could be re-created within the XGI system. Of course, because the XGI system does not replicate the entire functionality of the XQuery language, we would expect that not all queries could be replicated with XGI.

Of the 62 queries, the XGI system was able to exactly replicate 30 of these queries. In these cases, the XGI-generated query produced the same result as the saved XBrain query. In addition, we were able to partially replicate another 11 queries. In these cases, a large portion of the saved XBrain query was replicated, but the user would have to edit the XGI query before achieving the same results as the XBrain query. We consider these a partial success, as time is presumably saved by generating even just a portion of the desired query.

The principle reason why XGI cannot replicate all of these saved queries is that it is not designed to duplicate the entirety of the XQuery language. Table 1 is a partial listing of some of the XQuery features implemented and not implemented in XGI. The 21 XBrain queries that could not be replicated contained some of these un-implemented constructs, such as "if…then…", "concat", "count", and "union".

XQuery feature	in XGI?
Existential quantification	Yes
Conjunction	Yes
Breadth projection	Yes
Depth projection	Yes
New element	Yes
Join	Partially
Cartesian product	Partially
Nesting	No
Negation	No
Union	No
Arithmetic computations	No
Sorting	No

Table 1 XQuery features captured by XGI

In addition to testing XGI against an existing library of XQuery statements, we also conducted a preliminary user evaluation of XGI with an expert user of XQuery from our own group. The expert user installed the XGI system, learned the interface, and used XGI to create several queries. In general, the user found the interface easy to learn, and it functioned in an expected manner. However, the expert user did find that several features, such as the search ability and tooltip sub-menus, could have been better designed. Also, the user suggested that XGI provide feedback on the limitations of its query interface. For example, it should prevent users from creating an invalid query by validating the query against the source schema while the query is being constructed.

Discussion

Our preliminary evaluation and validation has helped us understand how well XGI satisfies our initial expectations, and how effective XGI is as a tool for generating XQuery queries. We have been successful in using XGI to expedite the creation of different types of XQuery statements. XGI can be used to browse the source schema, define the query schema, and visualize the query output graphically. XGI is also easy to install and set up on a centralized server, and the AJAX architecture means that individual users can work with XGI without any installation of a client application. Finally, the modular design of XGI allows for easy integration into other web-based applications.

Table I shows that XGI is unable to capture the full complexity and the variability of XQuery, which is reflected in the fact that we were unable to replicate 21 out of the 62 previously-generated CSM queries. Our belief is that even this degree of expressivity will be useful for researchers, especially when combined with manual editing. Indeed, most graphical querying tools use a simplified query language for this reason. This assertion needs to be tested for larger numbers of use cases.

However, it is likely that the queries generated by biomedical researchers will often become more complex than the current version of XGI can handle. Thus, in future work, we will need to extend XGI to implement additional complexity in the XQuery language, supporting operations such as nesting (hierarchical binding), aggregates, sorting, negation, filtering, arithmetic computations, and distributed query generation [9]. There are also some interface functionalities we would like to improve in XGI, such as support for adding multiple nodes to the query tree simultaneously, and implementing a "query-in-place" feature that can automatically validate the user's query schema.

These improvements should increase the usefulness and value of the XGI system. However, we must continually balance the usability of XGI with its expressivity. The more features we incorporate into XGI, the more complex it becomes, and this may ultimately erode its usability for end-users. In the end, XGI is limited by its visual querying paradigm: as a visual querying tool, XGI is designed to augment the query construction process, and it cannot completely replace expert informaticists who are experienced in the use of a complex query language such as XQuery. However, we argue that in bioinformatics settings, there is a strong need for tools to make the querying process simpler, and we believe that XGI is one such tool.

Acknowledgements

This work was funded by NIH grant DC02310. We thank Todd Detwiler for help with system evaluation.

References

1. World Wide Web Consortium. XQuery. http://www.w3.org/TR/xquery; 2001.
2. Structural Informatics Group. The University of Washington Integrated Brain Project. http://sig.biostr.washington.edu/projects/brain/; 2007.
3. Brinkley JF, Jakobovits RM, Poliakov AV, Martin RF, Gibson ER, Corina DM, Ojemann GA. An experiment management system for cortical stimulation mapping data. In: Society for Neuroscience Annual Meeting. San Diego; 2004. p. 1032.12. http://bmap.biostr.washington.edu/.
4. Tang Z, Kadiyska Y, Li H, Suciu D, Brinkley JF. Dynamic XML-based exchange of relational data: application to the Human Brain Project. In: Proceedings, Annual Fall Symposium of the American Medical Informatics Association. Washington, D.C.; 2003. p. 649-653. http://sigpubs.biostr.washington.edu/archive/00000141/.
5. Zloof M. Query by Example. In: Proceedings of the National Computer Conference, AFIPS; 1975. p. 431-438.
6. Li X. XGI: A graphical interface for XQuery creation and XML schema visualization [Masters]. Seattle: University of Washington; 2006.http://sigpubs.biostr.washington.edu/archive/00000198/
7. Petropoulos M, Papakonstantinou Y, Vassalos V. Graphical query interfaces for semistructured data: the QURSED system. ACM Transactions on Internet Technology 2005;5(2):390-438.
8. Braga D, Campi A, Ceri S. XQBE (XQuery By Example): A visual interface to the standard XML query language. ACM Transactions on Database Systems 2005;30(2):398-443.
9. Bales N, Brinkley J, Lee ES, Mathur S, Re C, Suciu D. A framework for XML-based integration of data, visualization and analysis in a biomedical domain. In: Proceedings, Third International XML Database Symposium (XSym 2005). Trondheim, Norway; 2005. p. 207-221. http://sigpubs.biostr.washington.edu/archive/00000178/.