Using XQuery to process XML documents for the APEX platform

Miguel Obregon

October 3, 2012

1 Goal

The main goal of this report is to explain the implementation of XQuery language to process XML documents that is used to create experiment files in the APEX 3 platform (Francart et al., 2008)[1]. In this specific case, we deal with the Interactive module of APEX 3.

2 Introduction

APEX 3 uses Xerces-C++ XML Parser [2] to parse XML documents and XALAN [3] that transforms XML documents into HTML, text, or other XML document types. For the previous implementation XALAN is used to generate XML document types. The experiment file is used to define the experiments in APEX 3. It contains different nodes that defines the experiment. From these elements, the interactive element allows the experimenter to modify certain aspects of the experiment file right before the experiment is started using a GUI [4].

3 XQuery language

XQuery (version 1.0) is a language used to perform queries in XML documents. XPath (version 2.0) and XQuery share the same data model and support the same functions and operators. This language allows us to select the XML data elements of interest, reorganize, and then return the result using a specific structure. One of the main differences between XQuery and XSLT is how the document is processed. XSLT implementations are generally optimized for transform entire documents. In addition, XQuery can establish queries from a selection of documents. XSLT 2.0 has the same characteristic. However, XSLT processors are not particularly optimized for this case. Comparing with XALAN/XPath, both libraries can compile this type of query. However, XALAN/XPath support DOM transformations. QT/XQuery cannot be implemented using the DOM API from QXml module. Thus, in order to read values from child nodes in a XML document, we need to call each child node name separately (an example of this case is shown in Listing 5). In the same way, we cannot modify a XML document since DOM API does not support Qt/XPath.
A XQuery expression is composed of two main parts: a *prolog* and a *body*. The query prolog is an optional section that is used to declare namespaces, variables, functions, and

to import schemas. The body query contains a sequence of one or more expressions that represents the query for the XML document. A example of a basic query is shown in Listing 1.

```
declare variable $input external;
for $x in doc($input)//devices/device/gain/
return $x
```

Listing 1: XQuery example

In this example the prolog contains the definition of the variable *$input* which is used to pass as argument the name of the XML document. The body contains a query that is executed over the XML code shown in Listing 2.

```
<?xml version="1.0" encoding="UTF-8"?>
<apex:apex xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://med.kuleuven.be/exporl/apex/3.0.2/experiment
https://gilbert.med.kuleuven.be/apex/schemas/3.0.2/experiment.xsd"
xmlns:apex="http://med.kuleuven.be/exporl/apex/3.0.2/experiment"
version="1">

  <devices>
    <device id="wavdevice" xsi:type="apex:wavDeviceType">
      ...
      <gain>0</gain>
      ...
    </device>
  </devices>

</apex:apex>
```

Listing 2: XML code used in Listing 1

This XML code represents the interactive module that defines the experiment files. The result of the query for this example is shown in Listing 3 In order to return the result without taking into account the namespace defined in the XML document, we use the path `//devices/device/gain/`. The symbol `//` selects nodes in the document from the current node (in this case the root node) that match the selection no matter where they are [5].

```
<gain>0</gain>
```

Listing 3: XQuery result

In order to get rid of the nodes in the result, we add the expression `text()` in the path. Thus, the final path will be `-//devices/device/gain/text()`.

4 XQuery in the experiment file

The value of the attribute *expression* (Interactive element) in the experimental file is interpreted by the XALAN library to obtain the path to a value of the same experimental file. Replacing these values with queries using XQuery language we can obtain the same result. The final code for the Interactive element is shown in Listing 4.

```
<?xml version="1.0" encoding="UTF-8"?>
<apex:apex xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://med.kuleuven.be/exporl/apex/3.0.2/experiment
https://gilbert.med.kuleuven.be/apex/schemas/3.0.2/experiment.xsd"
xmlns:apex="http://med.kuleuven.be/exporl/apex/3.0.2/experiment"
version="1">

<interactive>
 ...
<entry type="int" description="output gain"
 expression="doc($input)//devices/device/gain/text()"
 default="0"/>
 ...
</interactive>

</apex:apex>
```

Listing 4: XQuery implementation in Interactive element

The value of the attribute *expression* will be read by the C++ code in order to write a new experimental file with the results chosen in the GUI.

5 QXml module

In order to implement queries using XQuery language in C++, *QXmlQuery* library from Qt libraries will be used. In addition, we have to consider the process of reading and writing XML documents using Qt libraries. There are three ways to manipulate documents using QtXml module [6]. For this implementation, the Document Object Model (DOM) API will be used. This API converts a XML document into a tree structure, which the application can then implement queries.
Listing 5 shows a member that reads a XML document using this API.

```
#include <QFile>
#include <QMessageBox>
#include <QBuffer>
#include <QXmlQuery>
#include <QXmlFormatter>
#include <QDomDocument>

void class_name::readXML(){

  /*We select the document type to be parsed and the file content
  is assigned to it*/
  QDomDocument document;

  /*We select the file to be parsed*/
  QFile file("file_name.xml");

  /*Error string*/
```

```
18  QString errorMsg;
19
20  /*Error codes*/
21  int errorLine;
22  int errorColumn;
23
24  /*We define the list to collect the values from XML file*/
25  listDescription = new QStringList;
26  listDefault  = new QStringList;
27  listExpression = new QStringList;
28
29  QString text;
30
31  /*In case the file cannot be opened, we show an error message*/
32  if (!file.open(QIODevice::ReadOnly|QIODevice::Text)){
33    QMessageBox::critical(window, "QDomDocument",
34    "Could not open the file", QMessageBox::Ok);
35    return;
36  }
37
38  /*We parse the XML document from the file and sets it as the
39  content of the document*/
40  if (!document.setContent(&file, &errorMsg, &errorLine, &errorColumn)){
41    file.close();
42    return;
43  }
44  file.close();
45
46  /*Find the root element*/
47  QDomElement rootElement = document.documentElement();
48
49  if (rootElement.tagName() != "apex:apex")
50    return;
51
52  QDomElement child = rootElement.firstChildElement();
53
54  while (!child.isNull()){
55    //We get the name of the child node
56    text = text + child.nodeName() + "\n";
57
58    if(child.nodeName() == "interactive"){
59      QDomElement childElementEntry = child.firstChildElement();
60
61      while(!childElementEntry.isNull()){
62        listDescription->append(childElementEntry.attribute("description",""));
63        listDefault->append(childElementEntry.attribute("default",""));
64        listExpression->append(childElementEntry.attribute("expression",""));
```

```
65
66          childElementEntry = childElementEntry.nextSiblingElement();
67        }
68      }
69
70      //We go to the next node element
71      child = child.nextSiblingElement();
72    }
73 }
```

Listing 5: Reading using QTXml module (DOM API)

In order to read the experiment file to get the values for the GUI, first we open the file using the QFile library and assigning it to the variable *file*(line 15). Next, we have to verify if the file can be opened correctly using the member *open* from QFile (lines 32-36). In line 40 *setcontent* read the XML document and returns true if the content was successfully parsed. Since we need to extract an specific node from the XML document, we proceed as shown in lines 46-72. First, we define the root element using the *firstChildElement* member from QDomElement library. Next, using the same member recursively we can select the child nodes of that node. Finally, after selecting the *entry* node, the values of the attributes are stored in three different lists according to the attribute (description, default and expression).

In order to use XQuery language using the QXmlQuery library, we proceed as shown in Listing 6. We proceed in a similar way as explained in previous listing: First, the XML document is opened (lines 2-4). Next, a byte array is defined to store the result of the query (lines 6-8). The *query* variable is defined to use the members of QXmlQuery library (line 10). We bind the value *input* to the name of the XML document. This variable is linked with the prolog defined in Listing 1 that sets the same variable *input* to use as name of the XML document. The member *evaluate* has as only argument the query passed as string (*str*) and it will be used to set the query (line 13). Finally, the query is evaluate using *evaluateTo* (line 19) and the result is stored in the variable *resultEvaluation*.

```
1 void class_name::evaluate(const QString &str){
2
3   QFile sourceDocument;
4   sourceDocument.setFileName("file_name.xml");
5     sourceDocument.open(QIODevice::ReadOnly);
6
7   QByteArray outArray;
8     QBuffer buffer(&outArray);
9     buffer.open(QIODevice::ReadWrite);
10
11    QXmlQuery query;
12    query.bindVariable("input", &sourceDocument);
13  query.setQuery(str);
14
15  /*We check if the query is valid*/
16    if (!query.isValid())
```

```
17         return;
18
19     QXmlFormatter formatter(query, &buffer);
20     if (!query.evaluateTo(&formatter))
21         return;
22
23   buffer.close();
24   resultEvaluation = QString::fromUtf8(outArray.constData());
25 }
```

Listing 6: XQuery implementation using QXmlQuery library

6 Conclusion

In this file we explained XQuery language implementation using QTXml module. Examples for reading, parsing, and writing XML documents using this module has been implemented. The main drawback of this implementation is that we cannot use XPath to evaluate DOM API from QXml module. A possible solution is to use identity transform using XSLT or XQuery. The identity transform is a data transformation that copies the source data into the destination data without change [7]. However, it can be used to make local modifications in the XML document.

References

[1] T. Francart, A. van Wieringen, and J. Wouters, "Apex 3: a multi-purpose test platform for auditory psychophysical experiments," *Journal of Neuroscience Methods*, vol. 172, no. 2, pp. 283–293, 2008.

[2] Apache Software Foundation, "Xerces-C++ XML Parser." `http://xerces.apache.org/xerces-c/`, September 2012.

[3] Apache Software Foundation, "Apache Xalan Project." `http://xalan.apache.org/`, September 2012.

[4] A. Van Wieringen, T. Francart, and L. Van Deun, "APEX 3 User Manual," September 2012.

[5] W3Schools, "XPath Tutorial." `http://www.w3schools.com/xpath/xpath_syntax.asp`, September 2012.

[6] J. Blanchette and M. Summerfield, "C++ GUI Programming witrh Qt4: XML." `http://www.informit.com/articles/article.aspx?p=1405553`, November 2009.

[7] W. T. F. Encyclopedia, "Identity Transform." `http://en.wikipedia.org/wiki/Identity_transform`, April 2012.