Combining UML, XML and relational database technologies – the best of all worlds for robust linguistic databases

Larry S. Hayashi
SIL International
7500 W. Camp Wisdom Rd.
Dallas, TX USA 75236
larry_hayashi@sil.org

John Hatton
SIL International
7500 W. Camp Wisdom Rd.
Dallas, TX USA 75236
john_hatton@sil.org

Abstract

This paper describes aspects of the data modeling, data storage, and retrieval techniques we are using as we develop the FieldWorks suite of applications for linguistic and anthropological research. Object-oriented analysis is used to create the data models. The models, their classes and attributes are captured using the Unified Modeling Language (UML). The modeling tool that we are using stores this information in an XML document that adheres to a developing standard known as the XML Metadata Interchange format (XMI). Adherence to the standard allows other groups to easily use our modeling work and because the format is XML, we can derive a number of other useful documents using standard XSL transformations. These documents include 1) a DTD for validating data for import, 2) HTML documentation of diagrams and classes, and 3) a database schema. The latter is used to generate SQL statements to create a relational database. From the database schema we can also generate an SQL-to-XML mapping schema. When used with SQL Server 2000 (or MSDE), the database can be queried using XPath rather than SQL and data can be output and input using XML. Thus the Fieldworks development process benefits from both the maturity of its relational database engine and the productivity of XML technologies.

With this XML in/out capability, the developer does not need to translate between object-oriented data and relational representation. The result will be, hopefully, reduced development time. Another further implication is the potential for an increased interoperability between tools of different developers. Mapping schemas could be created that allow FieldWorks to easily produce and transfer data according to standard DTDs (for example, for lexicons or standard interlinear text). Data could then be shared among different tools –in much the same way that XMI allows UML data to be used in different modeling tools.

Introduction

In our current development project, called FieldWorks, we are using the following technologies and methodologies: object-oriented analysis, the Unified Modeling Language, XML, XSL transformations, XSchema[1] and SQL Server 2000. This paper examines how we are combining these, the problems we have run into and the real and potential advantages that this strategy provides for linguistic databases.

1 FieldWorks Philosophy

FieldWorks is one of SIL International's software development projects geared to help field researchers gather language and culture data. FieldWorks is based on the same computational philosophy used in the Computing Environment for Linguistic, Literary and Ethnographic Research, henceforth

[1] For XML, XSL Transforms and Xschema specifications, refer to http://www.w3c.org.

CELLAR (Simons, 1998). The CELLAR environment was first implemented in a product called LinguaLinks[2], which FieldWorks will eventually replace.

1.1 The nature of language and culture data

CELLAR was designed on the premise that language and culture data is highly integrated. For example, recorded vernacular stories provide material for a variety of cultural and linguistic analyses. If the researcher wants to transcribe and parse the text, the system will use morphological information from the lexicon. Parsed texts may be used in a variety of discourse analyses. The researcher might also want to employ the lexicon and morphological grammar to check "spelling" in non-elicited translated material.

2 FieldWorks data models

The FieldWorks data models are derived using object-oriented analysis methodology (Rumbaugh et al. 1991). OOA is particularly well suited for the hierarchical structures found in linguistic data. We currently capture these models and classes using a notation called the Unified Modeling Language (UML).

UML has become the industry standard for specifying, visualizing, and documenting the artifacts of software systems.[3] It is similar to other modeling languages, especially those developed by Booch, Jacobson or Rumbaugh, as these three were the primary creators of UML. Adopted as a standard by the Object Management Group[4] in late 1997, it is now supported by a number of data modeling tools. These tools allow the software analyst to simultaneously create visual models while creating a repository of data classes, attributes and their documentation. Because the repository of classes is normalized, the user can use the same class in a number of visual models. This is very helpful when modeling integrated linguistic data structures. For example, a particular linguistic class for "text transcription" might be used in an acoustic analysis model, a lexical data model, a discourse model, etc. The software analyst can customize the class for each of the tasks and models that the class is involved in, knowing that this change will be reflected throughout all diagrams. The analyst is thus better able to see the ramifications of his model changes.

A UML tool[5] has been indispensable to keeping documentation, diagrams and the database as up-to-date as possible. Our development process previously maintained three sets of files that had to be kept in sync via human intervention. We had diagram files, class definitions for generating the database, and a separate MS-Word document for describing the classes and their attributes. These have been unified into one source file in the UML tool that we use.

2.1 Storing UML models as XMI

Some UML applications have the capability to store their data in the XML Metadata Interchange format (XMI)[6] – an emerging standard for storing UML and other types of meta-data. There are a number of benefits to using this format. Research groups can easily exchange their data modeling work if they are using XMI conformant applications.[7] More importantly, because XMI is an instantiation of XML, standard XSL transformations can be applied on this meta-data to easily create other documents. The UML tool that we are using employs XSLT to create HTML documentation, including visual models[8] with hyper-links to

[2] http://www.ethnologue.com/lingualinks.asp

[3] Refer to http://www.uml-zone.com/umlfaq.asp for an excellent summary of UML.

[4] Refer to http://www.omg.org for more information about the Object Management Group.

[5] There is an excellent comparison of UML tools at http://www.objectsbydesign.com/tools/modeling_tools.html. The UML tool that we use, MagicDraw, is commercially available at http://www.magicdraw.com.

[6] XMI is a DTD for validating meta-data documents. For the actual specification, refer to http://www.omg.org/technology/xml/index.htm.

[7] Although classes and their attributes, relationships, etc. will transfer, the visual models will not as these are not yet part of the standard.

[8] The UML tool generates a PNG compressed graphic with an associated HTML <map>. The user can click on a class or association and get to the

relevant class documentation. Using other XSL transformations, we also generate:

(1) SQL statements that generate the FieldWorks database.
(2) an XML Schema that is used to validate data documents before importing them into the generated database.

In order to create the SQL statements in (1) above, we first create a simplified version of the XMI file. This simplified file, henceforth CELLAR XML, is a distillation of the verbose XMI file containing only the bare essentials to generate the database. The file is processed by a small SIL-developed executable that generates SQL statements.[9] These statements are then executed to create the database in Microsoft SQL Server 2000.

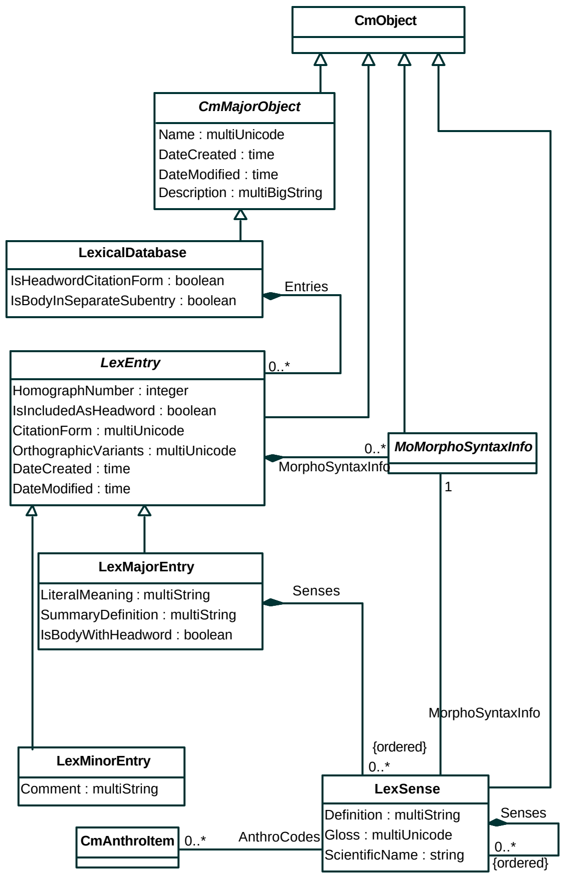
3 The FieldWorks database

Despite our preference to use a truly object-oriented database (such as LinguaLinks/Cellar which was programmed in object-oriented SmallTalk), we chose to use SQL Server 2000. We felt we needed to do this for a number of reasons. Our users were requesting multi-user capability, where two or more users can simultaneously work on the same linguistic database over a network (for example, two users simultaneously editing data in the same lexicon). Developing our own multi-user OO database proved to be a daunting task. We recognized that our small team of developers could not compare to the number of resources Microsoft poured into making SQL Server reliable under all sorts of conditions. Cost was also a factor. Commercial multi-user object-oriented databases were outside the price range of our users while the Desktop Edition of SQL Server was essentially free.[10] Thus we chose to use the SQL Server database – inexpensive, reliable, fast, and it provides network support.

3.1 OO models in a relational database

In moving to a relational DB, we did not want to lose the OO perspective of our data models because OO methodologies have proved so helpful to us in modeling linguistic data. Fortunately, we have been able to implement our OO models within a relational database in a way that supports inheritance and hierarchy.[11] Below, we see how a part of a FieldWorks OO model (3) is translated to the relational equivalent found in (4).

(3) FieldWorks Lexical Database OO Model



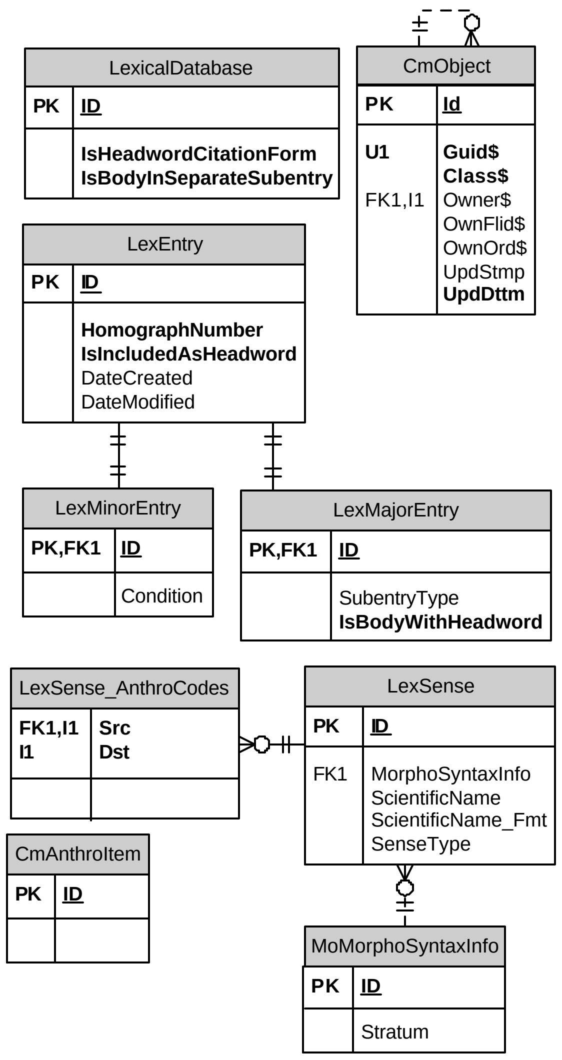
relevant documentation.

[9] Specifically, we generate a sequence of SQL insert statements which insert rows into two meta-tables, Class$ and Field$ (not shown). Triggers on these tables generate the tables for the model.

[10] MSDE is one of two database engine choices that can be used in MS-Access 2000 and later. It is essentially the same as SQL Server 2000 except that it only allows up to five concurrent users. SQL Server 2000 will scale to thousands of concurrent users. Programmers using Visual Studio or MS-Office can distribute MSDE for free to their user-base.

[11] Jacobsen et al. (1993) and Rumbaugh et al. (1991) discuss a number of principles to consider when doing this type of implementation.

(4) FieldWorks Relational Implementation



For every class in (3), both abstract and concrete[12], there is a corresponding table in (4) with the same name. Each of these tables contains a single row for each object in the system that is an instance of this class, either directly or through inheritance. These rows are tied together by the value of the ID column which is unique to each object. Note that even the highest level of abstraction, the CmObject in (3), has a corresponding table in (4).

For each attribute on a class that has a primitive signature (integer, boolean, time, etc.), the table that represents the class has a corresponding column. Our atomic associations (e.g. MorphoSyntaxInfo on LexSense) are directional and represented by a column on the source containing the ID of the destination object. Non-atomic associations (e.g. AnthroCodes on LexSense) are represented by a separate joiner table whose name combines the class name and property name (e.g. LexEntry_Senses). Owning relationships[13] (e.g. Senses are owned by LexMajorEntry) are represented in the CmObject table in a row keyed to the owned object. The ID of the owner is found in the Owner$ column, while the OwnFlid$ column contains a "field id" indicating which field of the owner is associated with this object[14]. For non-atomic relationships that are ordered, a sequence number is stored in OwnOrd$.

The signatures multiString and multiUnicode (e.g. Gloss: multiUnicode on LexSense in (3)) allow us to store data in more than one language. For example, we might want to gloss the sense of a lexical entry in English, French and a regional trade language. MultiUnicode and multiString signatures allow us to add as many different languages as the user deems necessary without having to create new attributes for each language in the OO model. This implementation separates the multilingual requirement of a field from the function or semantics of that field. MultiString signatures also allow the user to specify formatting (font, bold, italic, etc.) for spans of the string[15].

The SQL Server database contains a number of special meta-tables that are used for identifying the field names for aggregations and to capture the multiUnicode and multiString signatures. Combined, the SQL statements necessary to look at the data can be very complex. As a result, when the database generator builds the database, it also builds SQL views[16] that the

[12] Abstract class names in (3) are in italics.

[13] Our term for UML *composite aggregations*. An aggregation is a whole-part relationship between classes.

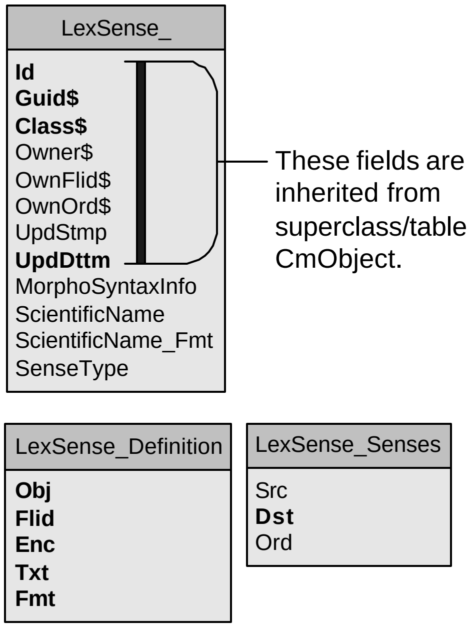
[14] Thus OwnFlid$ represents the name of the aggregation.

[15] For more information on the FieldWorks implementation of MultiUnicode and MultiString fields, refer to Thomson (2001).

[16] A SQL View is essentially a predefined query that

programmer can use to more easily access the data. The name of the view is a decorated version of the 'real' table's name. In (5) below, the default view for the class LexSense is shown and has the name *LexSense_* (note the trailing underscore). These views take into account inheritance and some of the specialized fields found in the meta-tables.

(5) Pre-built views on LexSense and LexSense_Definition (a multiString signature) and LexSense_Senses (an aggregation)



3.2 Using XML as a conduit for data

FieldWorks will eventually allow the user to parse textual data into morphemes based on a word grammar and an inventory of morphemes found in the lexicon. For years, some SIL field teams have been using AMPLE[17] (Weber et al., 1988), an SIL-developed morphological parser. We have been working on extending the linguistic capabilities of this parser in addition to adding XML in/out capabilities (its native data transfer format is SIL standard format[18]).

As in many other database applications, XML is becoming a standard way to represent data coming out of a database. This format makes an excellent conduit because it represents both the mark-up for the data elements (unlike raw, tab and comma-delimited formats) and the data itself. In addition, the data contained within the elements can be easily validated against a DTD or schema before moving on to other processes.

In FieldWorks we employ XML as a conduit between the SQL Server database and the extended AMPLE parser. The following sections describe how we can get XML data out of SQL Server 2000.

3.3 XML Functionality in SQL Server 2000

Because XML is becoming a standard way to deliver data on the web, most of the major relational databases now have built in XML functionality. SQL Server 2000 supports a number of methods to return XML results. For example, the simple addition of some XML keywords to the SQL statement in (6) will return results like those found in (7).

(6)
```
SELECT ID, MorphoSyntaxInfo
FROM LexSense
FOR XML AUTO
```

(7)
```
<LexSense ID="1563"
MorphoSyntaxInfo="1561"/>
<LexSense ID="1567"
MorphoSyntaxInfo="1562"/> …
```

SQL Server also provides a mechanism for a developer to output XML according to a particular schema. One advantage is that the output does not have to match the table and field names found in the database. Imagine that we want to deliver data from FieldWorks to some other tool that can use XML input that conforms to a particular DTD. SQL Server allows the developer to build these queries and return the results directly as XML.[19] We employ one of

can be used in other queries.

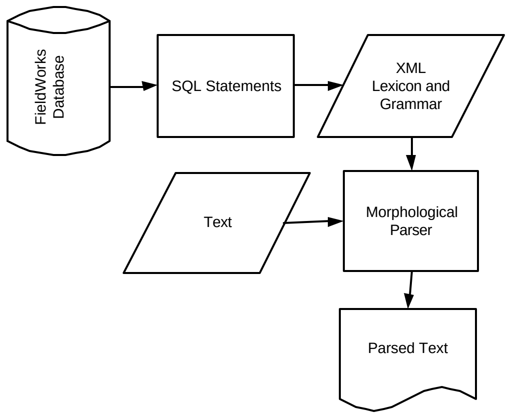
[17] AMPLE is an acronym for A Morphological Parser for Linguistic Exploration.

[18] SIL Standard format is a text file which delimits fields using a backslash followed by a two or three letter field code followed by a space followed by data (e.g. \lx iguana). Data is followed by a paragraph return and another field. Records are delimited by a key field.

[19] Techniques for reshaping XML data in SQL Server

these techniques to deliver morphological data as XML from the FieldWorks database to a parser using the process outlined in (8). This XML data conforms to the DTD for data that the parser is expecting. The words of the *Text* are then parsed using this information and *Parsed Text* is output as a file.

(8) Delivering XML data to the parser



If standard XML interchange formats are developed for general purpose linguistic annotation and lexical database tools, the FieldWorks database could deliver appropriately marked up data to the tool using this XML mechanism.

3.4 XML Views in SQL Server 2000

As described in 3.1, the representation of our object model in the SQL Server database requires the developer to create complex SQL statements to get at the data – which of course, means that the developer needs to have a good handle on SQL. As Williams et al. (2000) asks "Wouldn't it be great if we could query the database as though it were an XML document using XPath and other XML query languages?" As XML becomes a prevalent standard, more developers and users will become familiar with it. And wouldn't it be great if the XML document better reflected the object structure of the original object model so that we could query in an object-oriented manner? The XML Views feature in SQL Server 2000 provides us with this functionality.

are covered in Williams et al. (2000).

Using XML Views, we feed SQL Server an XSD (XML Schema Definition) that defines what we want the XML output to look like. In addition to this, we define supplementary SQL annotations for each element and attribute in the schema. These annotations instruct SQL Server how to retrieve the corresponding data from the database as in (9) below (arrows ➔ identify placement of SQL annotations). For each element, a sql:relation[20] defines the corresponding table or SQL view to use. Where a join is required between tables, a sql:relationship defines the parent table and its key along with the child table and the matching child key. The sql:relation attribute on an element definition ties the element to a table in the database. Attribute definitions include the sql:field attribute to indicate the corresponding column.

(9) XML schema with SQL annotations

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema
   xmlns:sql="urn:schemas-microsoft-com:mapping-
schema"
   xmlns:xs="http://www.w3.org/2001/XMLSchema"
   elementFormDefault="qualified"
   attributeFormDefault="unqualified">
 <xs:element
      name="LexicalDatabase"
  ➔  sql:relation="LexicalDatabase_">
  <xs:complexType>
   <xs:sequence>
    <xs:element
         name="Entries"
     ➔  sql:is-constant="1">               LexMajorEntry
     <xs:complexType>                       is referenced to
      <xs:sequence>                         element
       <xs:element                          definition
         ref="LexMajorEntry"                below.
         maxOccurs="unbounded"
     ➔  sql:relation="LexMajorEntry_">
        <xs:annotation>
         <xs:appinfo>
       ➔<sql:relationship
                parent="LexicalDatabase_"
                parent-key="ID"
                child="LexMajorEntry_"
                child-key="Owner$"/>
         </xs:appinfo>
        </xs:annotation>
       </xs:element>
      </xs:sequence>
```

[20] The sql:relation, sql:relationship and sql:field supplementary schema annotations are not general standards, that is to say that they are only used in Microsoft SQL Server implementations.

```
      </xs:complexType>
     </xs:element>
    </xs:sequence>
    <xs:attribute
        name="id"
        type="xs:byte"
        use="required"
    ➔ sql:field="ID"/>
   </xs:complexType>
  </xs:element>
  <xs:element
        name="LexMajorEntry"
    ➔ sql:relation="LexMajorEntry_">
   <xs:complexType>
    <xs:sequence>
     <xs:element
           name="Senses"
       ➔ sql:is-constant="1">
      <xs:complexType>
       <xs:sequence>
        <xs:element
              ref="LexSense"
              maxOccurs="unbounded"
          ➔ sql:relation="LexSense_">
         <xs:annotation>
          <xs:appinfo>
         ➔<sql:relationship
                 parent="LexMajorEntry_"
                 parent-key="ID"
                 child="LexSense_"
                 child-key="Owner$"/>
          </xs:appinfo>
         </xs:annotation>
        </xs:element>
       </xs:sequence>
      </xs:complexType>
     </xs:element>
    </xs:sequence>
    <xs:attribute
        name="id"
        type="xs:byte"
        use="required"
    ➔ sql:field="ID"/>
   </xs:complexType>
  </xs:element>
<xs:element name="LexSense" sql:relation="LexSense_"
sql:key-fields="Id" sql:max-depth="2">
  <xs:complexType>
   <xs:sequence>
  ...
    <xs:element
        name="Senses"
    ➔ sql:is-constant="1">
     <xs:complexType>
      <xs:sequence>
       <xs:element
            ref="LexSense"
            maxOccurs="unbounded"
        ➔ sql:relation="LexSense_">
        <xs:annotation>
         <xs:appinfo>
        ➔<sql:relationship
              parent="LexSense_"
              parent-key="ID"
              child="LexSense_"
              child-key="Owner$"/>
         </xs:appinfo>
        </xs:annotation>
       </xs:element>
      </xs:sequence>
     </xs:complexType>
    </xs:element>
   </xs:sequence> …
  </xs:complexType>
</xs:element>…
</xs:schema>
```

LexMajorEntry element

Recursive depth specificaion.

Once SQL Server has been set up to allow XPATH queries[21], the developer can submit a query (e.g. in 10) and receive the results as an XML document that conforms to the schema.

(10) XPATH query and results

XPATH query:

```
LexicalDatabase/Entries/LexMajorEntry[@id="1578"]
```

Results:

```
<root>
 <LexMajorEntry id="1578">
  <Senses>
   <LexSense id="1582">
    <Definition
          enc="740664001"
          txt="An English definition would go here."/>
    <Gloss enc="-1240214295" txt="Spanish gloss"/>
    <Gloss enc="740664001" txt="English gloss"/>
    <AnthroCodes>
     <CmAnthroItem id="862"/>
    </AnthroCodes>
    <Senses/>
   </LexSense>
   <LexSense id="1583">
    <Definition
          enc="740664001"
          txt="This is a definition of sense 2."/>
    <AnthroCodes/>
    <Senses/>
   </LexSense>
  </Senses>
 </LexMajorEntry>
</root>
```

Note in the XSD in (9) that each class (e.g. LexMajorEntry) is defined as its own root element. UML associations and aggregations, such as Entries on LexicalDatabase, are only defined within the elements in which they occur. This allows us to create XPATH queries that search for all members of a class rather than

[21] The procedure for setting up XML Views in SQL Server can be found in Williams et al. (pp. 582-624 :2000). It is also available in the SQLXML 2.0 Web release documentation downloadable from http://msdn.microsoft.com.

having to navigate the entire hierarchy. For example, the query in (10) could also be done as

(11) LexMajorEntry[@id="1578"]

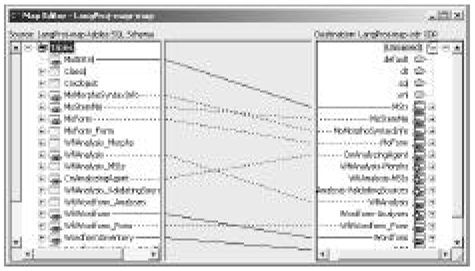
and return the same results found in (10).

The XML Views feature of SQL Server works by translating the XPATH query into a SQL statement. In our hybrid relational-OO implementation, this SQL statement can get rather large. Because we have root elements defined in the schema for each class, SQL Server can create a SQL statement based on that class and anything down the hierarchy rather than having to create the larger query necessary for navigating the entire hierarchy. The SQL statement is thus much smaller. Presumably, this should result in better performance.

Given an XSL transform along with the XPATH query, SQL Server can transform and deliver the results in presentation format (e.g. XHTML) or into another XML file that conforms to yet another schema.

One might think that defining the SQL annotations like those in (9) requires a lot of effort. Depending on the complexity of the database and the relationships between the tables, SQL Server can automatically determine the annotations. For more complex table relationships, the Microsoft SQL Server XML View Mapper[22] can be used to graphically draw relationships between XML elements and their relational table equivalents (12). The mapper lists tables and views on the left and schema elements on the right. Using a drag-and-drop interface, the user can establish the mapping relationships.

(12) SQL Server XML View Mapper



Due to its hybrid OO-relational nature, Fieldworks has many tables, and the relationships are complex. However, once the SQL annotations were determined for each type of relationship (there are approximately 15 different types of annotations), we were able to automatically transform the XMI file from our UML model into an XML Views schema with all necessary SQL annotations (we used an XSL transform). Here again, we found that we could readily exploit the XMI representation of our UML models to reduce the amount of human effort.

3.5 Current problems with XML Views

There are a few problems worth noting in using the XML Views strategy. For objects that have potentially recursive hierarchy (for example, in (3), LexSense can own another LexSense), it is necessary to explicitly state the maximum possible depth of the recursion. For every level of recursion, the SQL statement generated becomes larger, having to generate extra columns for the nested objects. As a result, SQL Server takes more time to create the SQL statement and then, subsequently, to execute it. For example, running a query on the XPATH "LexSense" takes less than 2 seconds on a control machine[23] when the depth is set to 2 compared to more than 15 seconds when set to 9 (this difference was noted on a data set that did not have recursive LexSenses).

Currently, we automatically generate one XML View schema that can query any part of the model. Because SQL Server is generating one large SQL statement for the XPATH statement,

[22] The Microsoft SQL Server XML View Mapper can be downloaded at : http://www.microsoft.com/sql/downloads/

[23] Pentium II 350 MHz with 384 MB RAM.

classes that have many attributes and associations will create very large SQL statements resulting in unacceptable performance.

Therefore, rather than create a single schema for the entire Fieldworks database, we find that we will need to create smaller, specialized ones. For example, for the purpose of feeding data to the AMPLE parser, we only need information relevant to a word grammar. We do not need all of the human generated prose that describes the grammatical objects. Thus, we plan to add UML tags to classes and their attributes which specify which schemas they should be included in. Then, our XSL transform which generates the schemas will be parameterized to include only those classes and attributes which are necessary for a particular task.

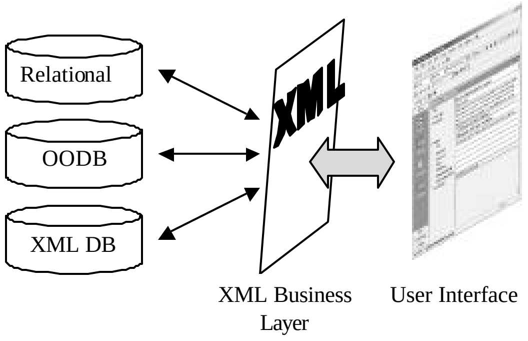
4 Future possibilities

4.1 XML as the business layer

SQL Server also has the capability to update data in the relational tables using the XML Views and XML *Updategrams* – in other words, it provides XML-in capabilities. We have not yet experimented with this functionality.

Eventually, OO databases will be more affordable and robust, multi-user XML databases will become available. Given that we have XML in and out capability, ideally, we should be building the FieldWorks interface in a way that it is database-independent and conduit-dependent. XML could be the conduit for the business layer (13). If we find it advantageous to switch databases in the future, we could do so with low cost to reprogramming the interface.

(13) XML as the business layer



4.2 Automatically generating user interfaces

In SIL, most of our linguistic field teams are using Shoebox[24], a linguistic database which operates on SIL standard format data[25]. In Shoebox, the user can specify a custom database model and immediately have a user-interface to populate the database. The data model definition is also the interface definition. In FieldWorks development so far, we have been designing interfaces specific to tasks that use the data. Although this should make for a friendlier user experience, the design and coding of such interfaces has slowed development considerably.

For data model testing and prototype development, it should be feasible to automatically generate user-interfaces for major objects. However, we do not want to create separate application programs for every class in the system. For example, it is sufficient to have a single application for editing lexical entries, their allomorphs, and their senses. Therefore, we may add another UML tag[26] to our model that would specify the editors to which a class would belong. Using another UML tag, we could also define what type of user-interface widget a particular attribute or association should use. Thus in the UML model, we would have everything necessary to automatically generate an application for viewing and modifying data. Then, using yet another XSL transform, we could generate an XML file that a FieldWorks user-interface generator could use.

[24] Refer to http://www.sil.org/computing/shoebox/

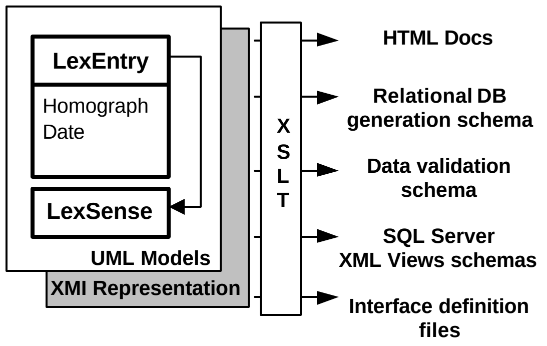
[25] Refer to footnote 18.

[26] In UML, tags can be added to any class or class attribute. They are used to specify information not accounted for by standard UML fields.

Conclusion

Stroustrup (1997) states that “constructing a useful mathematically-based model of an application area is one of the highest forms of analysis. Thus, providing a tool, language, framework, etc., that makes the result of such work available to thousands is a way for programmers and designers to escape the trap of becoming craftsmen of one-of-a-kind artifacts.” UML is an excellent example of such a language and framework. UML tools that make use of XMI provide even greater longevity and availability to the modeling work. XML is also such a language and framework. Because XMI is XML, we have been able to use standard XML tools and the XML functionality of SQL Server to easily derive a number of implementation specific products, as shown in (14).

(14) Derivatives of XMI in FieldWorks development



UML, XMI and XML provide a stable foundation for data modeling and software development. We expect our UML models to have longevity and we trust that the XMI representation will allow us to easily derive new functionality and better interface implementations as technology changes.

Acknowledgements

Our thanks go to the rest of the WordWorks development team (Andy Black, Randy Regnier, Mike Maxwell, Gary Simons), the FieldWorks development team, the volunteers who helped in the SQL database implementation (Shon Katzenberger, Paul Panek, Jonathan Richards and Valerie Hamm), Gary Duncanson of No Magic Inc., funding providers for both WordWorks and FieldWorks development, and organizers and funding providers of the IRCS Workshop including the University of Pennsylvania and the NSF.

References

Jacobsen, Ivar et al. (1993) *Object-Oriented Software Engineering – A Use Case Driven Approach*. ACM Press, Wokingham, England, pp. 269-283.

Rumbaugh, James et al. (1991) *Object-oriented modeling and design*. Prentice Hall, Englewood Cliffs, New Jersey, pp. 367-396.

Simons, Gary F. (1994) *Conceptual modeling versus visual modeling: a technological key to building consensus*. SIL. http://www.sil.org/cellar/ach94/ach94.html.

Simons, Gary F. (1998) *The nature of linguistic data and the requirements of a computing environment for linguistic research*. In “Using Computers in Linguistics: a practical guide”, John M. Lawler and Helen Aristar Dry (eds.). London and New York: Routledge, pp. 10-25.

Stroustrup, Bjarne (1997) *The C++ Programming Language*, 3rd edition. Addision Wesley. Reading, Massachusetts, p. 731.

Thomson, John (2001) *Representing multilingual text in memory in a relational database* in these Proceedings of the IRCS Workshop on Linguistic Databases. University of Pennsylvania, Philadelphia, USA.

Weber, David J. et al. (1988) *AMPLE: A tool for exploring morphology*. SIL International, Dallas, USA.

Williams, Kevin et al. (2000) *Professional XML Databases*. Wrox Pres Ltd., Birmingham, UK.

Additional SIL Computing Resources

FieldWorks Development Web Site: http://fieldworks.sil.org

SIL Computing website: http://www.sil.org/computing

SIL Language Software products including fonts, Shoebox, LinguaLinks, Ethnologue: http://www.ethnologue.com