CELLAR: A Data Modeling System for Linguistic Annotation

Gary F. Simons

SIL International
7500 W. Camp Wisdom Rd. Dallas, TX 75236, U.S.A.
gary_simons@sil.org

Abstract

CELLAR is not a particular annotation schema, but is a system for expressing and building annotation schemas. The paper illustrates how an annotation schema is expressed as an XML document that defines classes of objects, their properties, and the relationships between objects. The schema is then implemented via automatic conversion to a relational database schema and an XML DTD for data import and export.

Requirements for Linguistic Data Modeling

CELLAR[1] is a data modeling system that was built specifically for the purpose of linguistic annotation.[2] It was designed to model the following five fundamental aspects of the nature of linguistic data:[3]

1. The data in a text unfold sequentially; the data model must therefore be able to represent text in proper sequence.
2. The data are hierarchically structured; the data model must therefore be able to express hierarchical structures of arbitrary depth.
3. The data elements bear information in many simultaneous dimensions; the data model must therefore be able to annotate data objects with many simultaneous properties.
4. The data are highly interrelated; the data model must therefore be able to encode associative links between related pieces of data.
5. The data are multilingual; the data model must therefore be able to keep track of what language each datum is in.

Conceptual Modeling in CELLAR

CELLAR is not a particular annotation schema, but is a system for expressing and building annotation schemas. A particular annotation schema is called a *conceptual model* and is expressed as an XML[4] document which defines classes of objects, their properties, and constraints on the values of properties and the relationships between objects. A simplified version of the DTD for expressing conceptual models is given on the next page.

Modeling begins by identifying the classes of things in the world being modeled (i.e., the objects of object-oriented modeling or the entities of entity-relationship modeling). A class definition consists of documentation and a set of property definitions (which implement the third requirement above). Each class has a base class from which it also inherits properties; the ultimate base class is *CmObject*, for "conceptually modeled object." There are three kinds of properties: (1) *Owning properties* implement the part-whole relationships entailed by the second requirement that data are hierarchically structured. The cardinality

---

[1] The acronym is for "Computing Environment for Linguistic, Literary, and Anthropological Research". (See http://www.sil.org/cellar/ for more information.) CELLAR has been a team effort over the years and I am thus indebted to a host of colleagues. The leader of the development team has been John Thomson. Others who have had a major role in designing the things described in this paper are Shon Katzenberger, Stephen McConnel, and Ken Zook.

[2] "Linguistic annotation," by Steven Bird and Mark Liberman, Linguistic Data Consortium (2000). http://www.ldc.upenn.edu/annotation/.

[3] "The nature of linguistic data and the requirements of a computing environment for linguistic research," by Gary F. Simons. In *Using Computers in Linguistics: a practical guide*, edited by John Lawler and Helen Aristar Dry, pages 10-25. London and New York: Routledge (1998). An earlier version is available at http://www.sil.org/computing/computing_environment.html.

[4] XML is the "Extensible Markup Language"; see http://www.w3.org/TR/REC-xml for the specification and http://www.oasis-open.org/cover/ for a host of related resources.

```
<!-- Document Type Definition for a CELLAR Conceptual Model Definition
     Simplified version. Copyright (c) 2000 SIL International. -->

<!ELEMENT conceptualModel (comment?, class+)>
<!ATTLIST conceptualModel
     id    CDATA #REQUIRED><!-- id: The name identifying this model -->
<!ELEMENT comment (#PCDATA)>

<!-- ********** Class definitions ********** -->
<!ELEMENT class (comment?, descr?, props?) >
<!ATTLIST class
     id        ID      #REQUIRED
     abstract (true | false) "false"
     base      IDREF   #REQUIRED >
<!-- id:       name that is unique among all classes in model -->
<!-- abstract: only a superclass (true), or, has instances (false) -->
<!-- base:     id of base (super) class from which this inherits -->

<!ELEMENT props   (owning | rel | basic )*>

<!-- ********** Owning property definitions ********** -->
<!ELEMENT owning (comment?, descr?)>
<!ATTLIST owning
     id        CDATA          #REQUIRED
     card     (atomic | seq | col)   "atomic"
     sig       IDREF          #REQUIRED >
<!-- id:       non-unique among all attributes in all classes -->
<!-- card:     cardinality as single, sequence, or collection -->
<!-- sig:      name of signature class -->

<!-- ********** Relationship property definitions ********** -->
<!ELEMENT rel (comment?, descr?)>
<!ATTLIST rel
     id        CDATA          #REQUIRED
     card     (atomic | seq | col)   "atomic"
     sig       IDREF          #REQUIRED >
<!-- id:       non-unique among all attributes in all classes -->
<!-- card:     cardinality as single, sequence, or collection -->
<!-- sig:      name of signature class -->

<!-- ********** Basic Property definitions ********** -->
<!ELEMENT basic (comment?, descr?)>
<!ATTLIST basic
     id     CDATA        #REQUIRED
     sig   (Integer | Boolean | Time | GenDate | Numeric | Float |
            String | MultiString | Unicode | MultiUnicode |
            EncUnicode | Guid | Image ) "String" >
<!-- id:       non-unique among all attributes in all classes -->
<!-- sig:      primitive value type; default is String -->

<!-- ********** Documentation ********** -->
<!ELEMENT descr  (submod*)>            <!-- Description -->
<!ELEMENT submod (p | ul | ol)*>       <!-- Submodule -->
<!ATTLIST submod
   type (Summary | Description | Usage | Examples | Notes) #REQUIRED>
<!ELEMENT p      (#PCDATA | i | b )*>  <!-- Paragraph -->
<!ELEMENT i      (#PCDATA | b )*>      <!-- Italic -->
<!ELEMENT b      (#PCDATA | i )*>      <!-- Bold -->
<!ELEMENT ul     (li)*>                <!-- Bulleted list -->
<!ELEMENT ol     (li)*>                <!-- Numbered list -->
<!ELEMENT li     (#PCDATA | i | b )*>  <!-- List item -->
```

attribute can specify that the owned objects form a sequence, thus supporting the first requirement. (2) *Relationship properties* implement the fourth requirement, that the model must support associative links between related data objects. The signature attribute constrains what classes of objects can be the targets of a link. (3) *Basic properties* store primitive data values like numbers, strings, Booleans, dates, and binary images (such as for graphics or sound). The fifth requirement that data are multilingual is supported by the fact that the primitive type String allows spans of characters to be identified as to language, and MultiString and MultiUnicode support alternate renditions of the same string in multiple languages.

As a conceptual model is developed in XML, descriptions of the classes and properties are included right inside the definitions. As a result, an XSL stylesheet is able to render the conceptual model source code as hyperlinked documentation in a browser.

An Overview of the Implementation

CELLAR was first implemented in Smalltalk (beginning in 1990) as a stand-alone object-oriented database management system. The data store and the applications to manipulate those data were both modeled within the object-oriented database. This system was used to build the text analysis and lexicon management tools in the product named *LinguaLinks.*[5]

Now, ten years later, we are building a second-generation system based on a client-server model, which separates the implementation of the data store from that of the applications. This offers a number of advantages: (1) we are able to use an off-the-shelf product for the data server, (2) applications can be written in a variety of languages (including C++, Java, Visual Basic, and Dynamic HTML), (3) client and server can be on the same machine or can communicate across a network, and (4) the transaction processing capability of the data server opens up the possibility of safe multi-user write access to the same database.

A relational database engine is being used to implement the object store. Specifically, we are using Microsoft's SQL Server. In installations for use by individual linguists, we are able to use the Microsoft Data Engine[6] (MSDE) which is a freely distributable 1-to-5-user version of SQL Server.

Figure 1 gives an overview of the system implementation. The box represents the off-the-shelf database server. The ovals represent software that our project has implemented. The arrows represent data flows. The numbered items represent text files that serve as input or output. The dashed lines represent the relationship between a valid XML document (on the right) and the DTD to which it conforms (on the left).

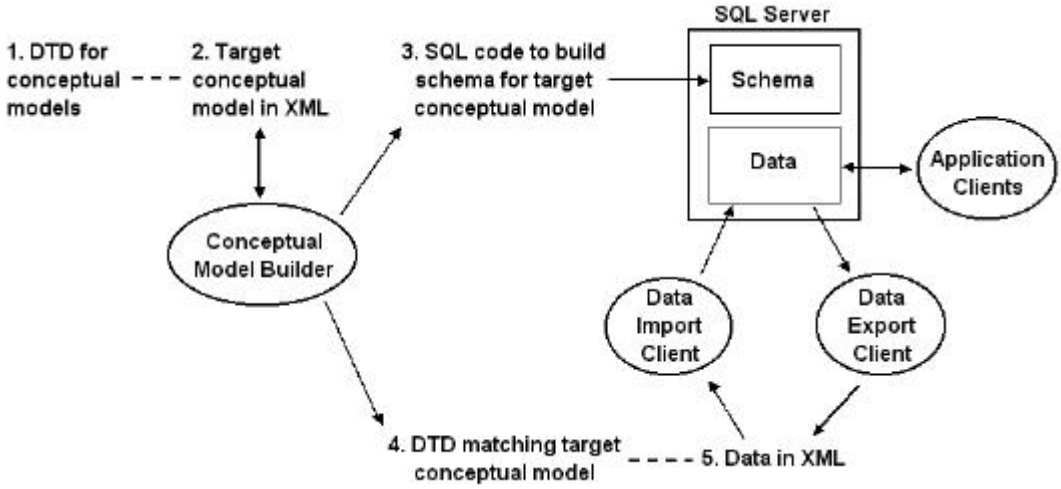

Figure 1. Overview of the CELLAR system

Building a CELLAR application begins with the Conceptual Model Builder. This has three components: the Class Manager (which gives a graphical user interface for creating and editing the XML representation [2] for the class definitions in a conceptual model in such a way that they conform to the DTD for conceptual models [1]), the Code Generator (which generates the SQL Data Definition Language code [3] that will implement the conceptual model as relational database schema), and the DTD Generator (which generates a

[5] See http://www.sil.org/lingualinks/LingTool.html.
[6] See http://msdn.microsoft.com/vstudio/msde/.

DTD [4] that defines the set of XML documents that can be loaded directly into that conceptual model). The SQL code [3] is then executed by SQL Server to build the schema for the database. Once the database schema has been created, the database can be populated with data by using the Data Import Client to read it from an XML document [5] that conforms to the DTD for this conceptual model [4] that was generated by the Conceptual Model Builder. The Data Export Client does the reverse, reading data from the database and outputting in XML that conforms to the same DTD. Multiple client application programs can be implemented in various languages to access and manipulate a given database; the client provides a user interface that is optimal for the task at hand, while the SQL data engine maintains data integrity (according to the constraints expressed in the conceptual model).

An Example

To illustrate the process in Figure 1, we will use the following simple entry from a dictionary of Sikaiana, a Polynesian language of the Solomon Islands:

> **manu** 1 [nf] a generic term for any animal of the air or on land, including large insects. 2 [n] a beast, as might be seen in a film or in a dream. 3 [na] a kite made from the leaf of the pandanus tree (*hala*).

The conceptual model that accounts for this entry has two classes: Entry and Sense. An Entry has two properties, the headword and the senses. Each Sense in turn has two properties, the part of speech and the definition. The part of speech label that appears in a sense is not a basic property of the sense; rather, it is a relationship property that points to a PartOfSpeech object which has properties like name, abbreviation, and description. The XML representation of the conceptual model for these two classes is as follows:

```
<class id="Entry" base="CmObject">
   <comment>A simple dictionary entry</comment>
   <props>
      <basic id="headword" sig="String"></basic>
      <owning id="senses" card="seq" sig="Sense"></owning>
   </props>
</class>

<class id="Sense" base="CmObject">
   <comment>A simple sense entry</comment>
   <props>
      <rel id="pos" sig="PartOfSpeech"></rel>
      <basic id="definition" sig="String"></basic>
   </props>
</class>
```

This representation is submitted to the Code Generator to generate SQL code for building the relational database schema (see next section) and the DTD Generator to generate a DTD for data import and export. The following is a somewhat simplified version of the DTD that would be generated for the above class definitions:

```
<!-- Generated elements -->
<!ELEMENT Entry ( headword | senses )* >
<!ATTLIST Entry id ID #IMPLIED >
<!ELEMENT headword ( Str )? >
<!ELEMENT senses ( Sense )* >

<!ELEMENT Sense ( pos | definition )* >
<!ATTLIST Sense id ID #IMPLIED >
<!ELEMENT pos ( Link )? >
<!ELEMENT definition ( Str )? >

<!-- Built-in elements -->
<!ELEMENT Link EMPTY >
<!ATTLIST Link target IDREF #REQUIRED >

<!ELEMENT Str (#PCDATA | Prop)* >
```

```
<!ELEMENT Prop EMPTY>
<!ATTLIST Prop
    enc          CDATA                #IMPLIED
    charStyle    CDATA                #IMPLIED
    italic       (off | on | invert)  #IMPLIED
    bold         (off | on | invert)  #IMPLIED >
```

Note that the elements representing objects (e.g. Entry and Sense) always have an optional attribute named *id* so that any object can be the target of a relationship (i.e., a Link.element). Note, too, that the elements representing properties always allow their contents to be missing. This is because we do not view missing data as an error since one typically cannot account for everything during the early stages of linguistic analysis.

The built-in String data type (represented by the element Str in the data import DTD) supports language encoding (via the *enc* attribute) as well as formatting (with user-defined styles via the *charStyle* attribute or with direct manipulation of about a dozen formatting properties of which only two are shown). For this example, ENG is the code used to represent a span of text in English, while SIK is used for text in Sikaiana. The following, then, is the encoding of the sample dictionary entry that the Data Import Client can automatically read into the SQL Server database built from the automatically generated SQL code. (Note the language switching in the last definition.)

```
<Entry>
  <headword><Str><prop enc="SIK"/>manu</Str></headword>
  <senses>
    <Sense>
      <pos><Link target="POS.nf"/></pos>
      <definition><Str><prop enc="ENG"/>a generic term for any animal of
         the air or on land, including large insects.</Str></definition>
    </Sense>
    <Sense>
      <pos><Link target="POS.n"/></pos>
      <definition><Str><prop enc="ENG"/>a beast, as might be seen in a
         film or in a dream.</Str></definition>
    </Sense>
    <Sense>
      <pos><Link target="POS.na"/></pos>
      <definition><Str><prop enc="ENG"/>a kite made from the leaf of the
         pandanus tree (<prop enc="SIK"/>hala<prop enc="ENG"/>).</Str>
      </definition>
    </Sense>
  </senses>
</Entry>
```

Notes on the SQL Implementation

The translation from conceptual model to relational database schema is not trivial. For instance, our current conceptual model for a linguistic field project involves 130 classes; the Code Generator produces 10,000 lines of SQL code to implement the corresponding relational database schema. A table is generated for each class in the model, with columns corresponding to properties defined on that class. In addition to having a row in the table for its class, an instance of an object also has a row in each of the tables for the superclasses in its inheritance hierarchy. Thus, for each non-abstract class in the model, a view is automatically generated that does a join on all the tables in which it participates.

Signature constraints on owning and relationship properties are translated into referential integrity constraints involving foreign keys. In addition, numerous stored procedures and triggers are created that insure other aspects of integrity when the database is updated.

The hierarchy embodied in owning properties is handled as an owner column in the CmObject table, since every object has an owner (except for a few root objects). An owning property is thus implemented as a view of CmObject that maps the relationship in reverse, that is, from owner to owned. Sequence is handled by adding a column that specifically records an object's position within the sequence. String values are implemented in two columns, one to hold the sequence of Unicode characters and the other to hold a binary representation of the language encoding and formatting information. A function library for application programmers provides user interface widgets that support this implementation of multilingual strings.