
*Computers and the Humanities* **33:** 85–101, 1999.



# Using Architectural Forms to Map TEI Data into an Object-Oriented Database

GARY F. SIMONS
*Summer Institute of Linguistics, 7500 W. Camp Wisdom Rd., Dallas, TX 75236, USA*
*E-mail: gary_simons@sil.org*

**Abstract.** This paper develops a solution to the problem of importing existing TEI data into an existing object-oriented database schema without changing the TEI data or the database schema. The solution is based on architectural processing. Two meta-DTDs are used, one to define the architectural forms for the object model and another to map the existing SGML data onto those forms. A full example using a critical text in TEI markup is developed.

## 1. Introduction

Much of the promise of SGML lies in the fact that descriptively marked up data can be freely interchanged between different sites and between different software systems. Indeed, this is part of the motivation behind the Text Encoding Initiative's *Guidelines for Electronic Text Encoding and Interchange* (Sperberg-McQueen and Burnard, 1994). Unfortunately, interchange of SGML data between software systems is not always so easy. This is because, while SGML-aware software understands the syntax of the markup, it may not necessarily understand the semantics. Architectural forms are a mechanism that SGML offers for helping to bridge this semantic gap.

This paper describes how architectural forms can be used to solve the particular problem of mapping SGML data into the semantic model of an object-oriented database. More specifically, the problem is to import existing SGML data into an existing object-oriented database schema without changing either the SGML data or the database schema. The target system is an object-oriented database system named CELLAR (for Computing Environment for Linguistic, Literary, and Anthropological Research – see SIL, 1998). The solution uses architectural processing to map the SGML data onto architectural forms that the CELLAR system can use to construct the appropriate structure of objects.

Section 2 of the paper discusses the basic differences between the SGML model of data and the object model, and illustrates why the mapping from SGML elements to objects is not a trivial one. Section 3 introduces architectural forms

and presents an architecture that maps SGML data onto objects. Section 4 explains how architectural processing can be used to automatically translate a document into its corresponding architectural forms. Section 5 gives a complete example of the automated process by which the SGML data are mapped onto the architectural DTD via an intermediate meta-DTD that encodes the mapping. The example used is that of a critical text edition encoded in TEI markup. Finally, section 6 discusses the results that have been achieved thus far.

## 2. The SGML Model Versus the Object Model

An SGML document type definition (DTD) has much in common with the conceptual model that results from an object-oriented analysis of a problem domain (Booch, 1994; Coad and Yourdon, 1991). Because of this, it is logical to conclude that SGML data should be particularly amenable to being imported into software that uses an object-oriented data model. This is not a trivial task, however, since there are some fundamental differences between the SGML model of data and the object model. In speaking of the "object model of data," I am referring specifically to the way object databases (Cattell, 1997) and conceptual modeling languages (Borgida, 1985) represent information. Such systems replace the simple instance variables of an object-oriented programming language with attributes that encapsulate integrity constraints and the semantics of relationships to other objects.

In SGML, the fundamental unit of data representation is the element. An element may have attributes, but these are limited to simple values that lack embedded structure. Complex structure is encoded by embedding other elements within the content of the original element.

In the object model, the fundamental unit of data representation is the object. Each object is either a primitive object that stores primitive data like a string or a number, or is a complex object that has attributes. The value of an attribute consists of embedded objects.

Thus elements recurse through content, while objects recurse through attributes. That is, complex structures are built in SGML by embedding all the elements for the parts as a single sequence in the content of the element for the whole. In the object model, by contrast, the objects for the parts are put into various attributes of the object for the whole. This fundamental difference in the way the two models handle complex structure explains why it is not possible to import SGML data directly into an object database schema without semantic mapping. Elsewhere (Simons, 1997b, 1997c) I have discussed in greater depth the difference between the two models and the inadequacy of a default mapping from the one to the other.

This fundamental difference means that the embedding of object within attribute within object (and so on) of an object database, must be encoded in SGML as element within element within element (and so on). Thus when mapping from SGML to objects we are confronted with a fundamental problem:

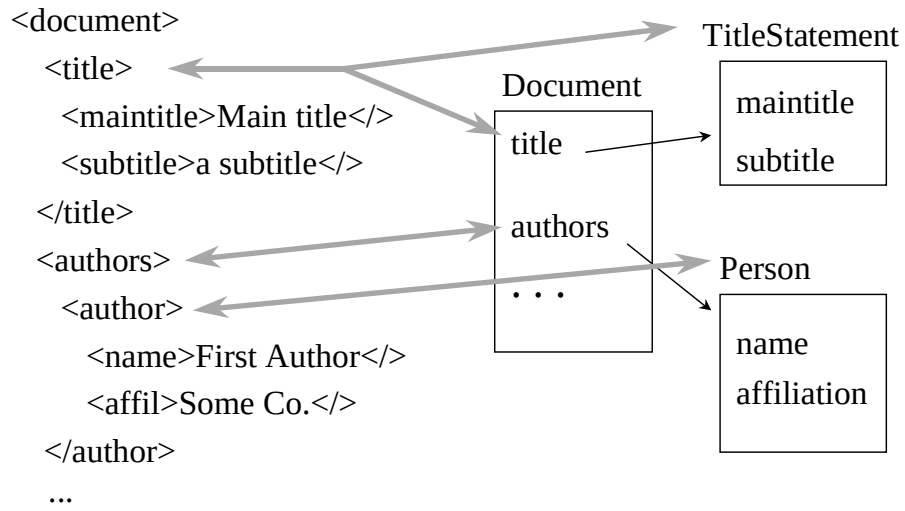


*Figure 1.* Mapping an SGML document to objects.

- Some elements encode an object.
- Some elements encode an attribute.
- Some elements simultaneously encode both.

Figure 1 illustrates this point. It shows a typical SGML document fragment on the left and a partial representation of a corresponding object structure on the right. The ⟨title⟩ element corresponds to both the *title* attribute of the Document object and the TitleStatement object which is its value. By contrast, the ⟨authors⟩ element corresponds to just the *authors* attribute, while the ⟨author⟩ element corresponds to just the Person object.

The three cases listed above are the most straightforward cases. An SGML element could also correspond to nothing in the object model, so that the element markup needs to be ignored and just its content processed. A single SGML element could correspond to two objects, one embedded within the other. An SGML attribute might correspond to an object's attribute. The basic challenge of importing SGML data into an object database is to determine which of these cases holds for each of the element types occurring in the data, and then to express formally how each maps onto the corresponding classes and attributes of the target database schema.

## 3. An Architecture for Mapping SGML Data into Objects

The HyTime standard (ISO, 1992; DeRose and Durand, 1994) first introduced the concept of architectural forms as a way to associate standardized semantics with elements in user-defined DTDs. HyTime is based on a fixed set of architectural forms. This approach to dealing with the semantics of markup was so successful

that the notion of architectural forms has been generalized. The generalized mechanism is one of the SGML Extended Facilities that was defined in an annex to the revised HyTime standard (ISO, 1997). Kimber (1997a) gives a tutorial introduction to the generalized architecture mechanism.

Now that architectural forms have been generalized, we can apply them to our problem of expressing the semantics of how SGML elements map onto the object model. A sampling of other problems to which architectures have been applied includes making documents accessible to people with print disabilities (Harbo and others, 1994), creating and managing literate programs (Kimber, 1997b), and labeling metadata in Internet resources (Kimber, 1997c). See Cover (1998) for an up-to-date listing of resources relating to architectural forms and their application.

In light of the increasing popularity of XML, it is worth noting that the HyTime standard has been amended to make it possible to use architectural forms in XML (Megginson, 1997). In *Structuring XML Documents*, Megginson (1998a) devotes three chapters to DTD design with architectural forms. He has also released an architectural engine for XML (Megginson, 1998b). One can also use an SGML parser like SP (Clark, 1997) to architecturally process an XML document provided that it is invoked with the SGML declaration for XML.

An *architecture* is like a semantic model. It is defined formally by a DTD. Syntactically, it is a normal DTD, but it is also known as a meta-DTD since it deals with information at a higher, more abstract level. Each element defined in an architectural DTD is called an *architectural form* and represents one of the semantic constructs of the architecture.

Figure 2 gives the DTD for the architecture used to map SGML data onto CELLAR's object model. There are two basic element forms in the architecture, ⟨object⟩ and ⟨attr⟩. Rather than having a third form for the case when an element corresponds to both an object and an attribute, this case is treated as being a mapping to an object, and the object form adds an architectural attribute to name the attribute it also maps to. A third form, ⟨ignore⟩, is used for the case when the SGML element does not correspond to anything in the target object model so the element content should be processed as though the start and end tags were not there. Note that the definition of the architecture is abridged for the sake of this presentation; the full definition is given elsewhere (Simons, 1997b, 1997c).

The easiest way to explain these forms is by example. Figure 3 shows a version of the illustrative document of Figure 1 which has been annotated to encode the mapping onto the object architecture. A special attribute, called the *architectural form attribute*, is added to each element. In this case we name the attribute "cellar" since it tells which architectural form in the CELLAR architecture the element corresponds to. For instance, the first element is annotated as: ⟨document cellar=object class=Document⟩. This indicates that the ⟨document⟩ element corresponds to an object in the CELLAR architecture; furthermore, it is an object of class Document. The ⟨title⟩ element is annotated as an object of class TitleStatement that the belongs in the *title* attribute of the parent (Document) object. The

```
<!-- CELLAR.DTD (abridged version)
     Meta-DTD of the CELLAR architecture for mapping
     SGML data into CELLAR's object model of data             -->

<!ENTITY % content "object | attr | ignore | #PCDATA"          >

<!--                                                          --
  -- OBJECT: the element corresponds to an object in CELLAR   --
  --                                                          -->
<!ELEMENT object - -  (%content;)*                             >
<!ATTLIST object
     class       -- Create this class of CELLAR object        --
                 CDATA #REQUIRED
     parentAttr  -- Put the object in this attr of its parent --
                 CDATA #IMPLIED
     contentAttr -- Put embedded objects in this attribute    --
                 CDATA #IMPLIED
     pcdataClass -- Create this class for embedded PCDATA     --
                 CDATA "String"
     encoding    -- Put embedded strings in this encoding     --
                 CDATA #IMPLIED
     id          -- A unique identifier for this object       --
                 ID    #IMPLIED
     attrName    -- Set this attribute of the object ...      --
                 CDATA #IMPLIED
     attrValue   -- ... to this value                         --
                 CDATA #IMPLIED
     attrType    -- The value is an IDREF or of named class   --
                 CDATA "String"

<!--                                                          --
  -- ATTR: the element corresponds to an attribute in CELLAR  --
  --                                                          -->
<!ELEMENT attr - -    (%content;)*                             >
<!ATTLIST attr
     contentAttr -- Put embedded objects in this attribute    --
                 CDATA #IMPLIED
     pcdataClass -- Create this class for embedded PCDATA     --
                 CDATA "String"
     encoding    -- Put embedded strings in this encoding     --
                 CDATA #IMPLIED                               >

<!--                                                          --
  -- IGNORE: the element corresponds to nothing in CELLAR;    --
  --         ignore it at this level, but process its content --
  --                                                          -->
<!ELEMENT ignore  - - (%content;)*                             >
```

*Figure 2.* Meta-DTD for CELLAR's object model.

⟨maintitle⟩ element corresponds to the attribute named *maintitle* in CELLAR. The remainder of the sample follows these same patterns.

Note that one requirement for using this technique is that the SGML data are marked up with at least the level of granularity that is needed by the object database. For instance, in the sample in Figure 3, the author's name maps into the object

```
<document cellar=object class=Document>
   <title cellar=object class=TitleStatement parentAttr=title>
      <maintitle cellar=attr contentAttr=maintitle>Main title</>
      <subtitle cellar=attr contentAttr=subtitle>a subtitle</>
   </title>
   <authors cellar=attr contentAttr=authors>
      <author cellar=object class=Person>
         <name cellar=attr contentAttr=name>First Author</>
         <affil cellar=attr contentAttr=affiliation>Some Co.</>
      </author>
   </authors>
   ...
</document>
```

*Figure 3.* Architecturally annotated SGML document corresponding to Figure 1.

database because the latter expects the full name as a string. If, on the other hand, the object database were to represent a name as an object with various attributes for the parts of a name, this technique could not produce the needed result from the SGML data since architectural processing cannot go into PCDATA content and parse special notations.

A related limitation of architectural processing is that it cannot insert an architectural element that has no corresponding element in the client document, not even when it could be inferred unambiguously from the client document's structure. For instance, if the DTD for the sample document in Figure 1 had no ⟨title⟩ element, but just put ⟨maintitle⟩ and ⟨subtitle⟩ directly within ⟨document⟩, then architectural processing would not be able to insert the architectural ⟨object class="TitleStatement"⟩ element that would be needed to achieve the proper mapping into the object model. In general, architectural processing cannot perform any transformations on the element structure of the client document beyond omitting certain elements. When the mapping to an object database requires major restructuring, this would need to be done as a preliminary step with a structure transformation tool.

## 4. Using Architectural Processing to Translate a Document

The architectural processing feature of an SGML parser is used to translate the elements of an input document into the corresponding elements of an architecture. The SGML parser reads an input document with its DTD (called the *client document* and the *client DTD*) and produces an output document that conforms to a different DTD (called the *architectural document* and the *architectural DTD*). Figure 4 gives a graphical overview of the process.

With the *nsgmls* parser from the SP package (Clark, 1997), architectural processing is invoked by giving the -A command line option. Following the -A is the name of the architecture to use. The name must be declared in an ARCBASE

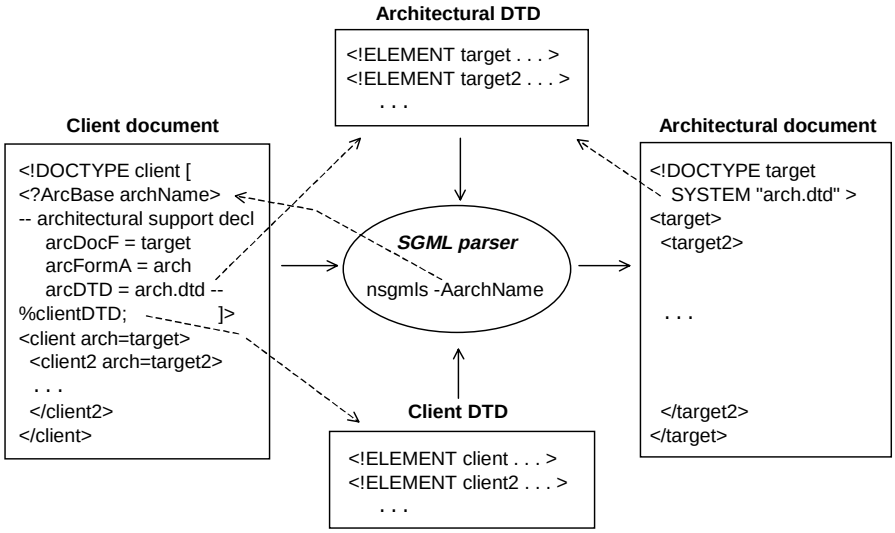


*Figure 4.* An overview of architectural processing.

processing instruction in the client document. Following this is the *architectural support declaration* that tells the parser how to process the architecture. (The notation is grossly simplified in the diagram.) This declaration specifies the DOCTYPE of the architectural document (arcDocF), the architectural DTD (arcDTD), and the architectural form attribute (arcFormA). In Figure 4, arch is specified as the architectural form attribute. Thus ⟨client arch=target⟩ means that the corresponding element in the architectural document is ⟨target⟩. When the parser is performing architectural processing, it not only translates the elements of the client document into the corresponding elements of the architecture, but also validates the architectural document being produced against the architectural DTD to ensure that the output is a valid instance of the architectural document type. Figure 5 shows the output of performing architectural processing on the sample document in Figure 3, where cellar is the architectural form attribute.

The process diagrammed in Figure 4 requires that the client document be already annotated with the values of the architectural form attribute and other architectural attributes. This, however, violates our basic requirement that the process for importing SGML data should not require that we change the SGML data file. This problem can be solved by performing architectural processing twice, first to add the architectural attributes to the client document and then to create the architectural document. The nsgmls parser can do this in one pass by specifying two -A options on the command line. Figure 6 illustrates this process.

In the first step in Figure 6, the architecture named mapping is invoked. It uses a *mapping DTD* to supply the architectural form attribute and any other architectural attributes for each element type in the client document. The mapping DTD consists of a series of ATTLIST declarations that supply the needed attributes. For instance,

```
<object class=Document>
   <object class=TitleStatement parentAttr=title>
      <attr contentAttr=maintitle pcdataClass=String>Main
title</attr>
      <attr contentAttr=subtitle pcdataClass=String>a
subtitle</attr>
   </object>
   <attr contentAttr=authors>
      <object class=Person>
         <attr contentAttr=name pcdataClass=String>First
Author</attr>
         <attr contentAttr=affiliation pcdataClass=String>Some
Co.</attr>
      </object>
   </attr>
   ...
</object>
```

*Figure 5.* Architectural document corresponding to Figure 3.

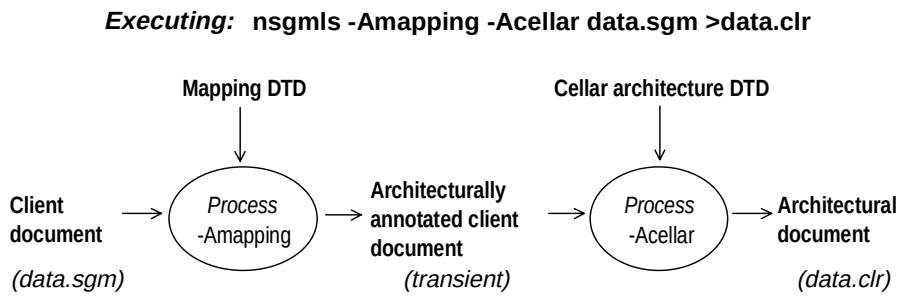


*Figure 6.* Two-stage architectural processing.

mapping the ⟨author⟩ element of Figure 1 into a Person object would be done with the following declaration:

⟨!ATTLIST author cellar NAME #FIXED object
class CDATA #FIXED Person ⟩

The output of the first application of architectural processing in Figure 6 is an architecturally annotated version of the client document. This document is transient in that it is never written to a file (though it can be written by running just nsgmls -Amapping). The architecture named cellar is then invoked with the transient document as input. The result is the corresponding document that conforms to the architecture for the CELLAR object model. The entire process is now exemplified in the next section.

## 5. From TEI Data File to Object Database: A Complete Example

As stated in the introduction, the goal of this work is to import existing SGML data into an existing object-oriented database schema without changing the SGML data or the database schema. This is done by means of a six-step process. The following six subsections take a complete example through these steps.

The SGML data file used in the example is given in Figure 7. It is a critical edition in TEI markup of a passage from the Second Epistle of Clement. Note that a significant portion of the content has been elided in the interest of brevity. (The Greek text is encoded in TLG beta code.) A fuller treatment of this sample text along with examples of what can be done with it in the CELLAR environment is given in an article published previously in this journal (Simons, 1997a).

### 5.1 Compare the original DTD and the CELLAR conceptual model

The first step in the process is to compare the DTD for the SGML data file with the conceptual model for the target database. The purpose is to determine what class or attribute in the object model corresponds to each SGML element and attribute. Figure 8 shows the original DTD for the sample TEI data file.

The conceptual model for the CELLAR objects and attributes into which we want to import the sample data file is diagrammed in Figure 9. The notation and the model are explained in Simons (1997a). Here suffice it to say that solid arrows mean "contains" and the dotted arrow means "holds pointers to."

### 5.2 Create the mapping DTD

The correspondence between the elements and attributes of the original DTD and the objects and attributes of the CELLAR conceptual model are formally expressed in the mapping DTD. It is a meta-DTD that consists primarily of ATTLIST declarations for the elements of the original DTD; they serve to set a value for the architectural form attribute and for the attributes of that form.

The mapping DTD for our example is given in Figure 10. The ⟨?ArcBase cellar⟩ processing instruction declares *cellar* to be the name of the base architecture. Following this is the architectural support declaration. It consists of a notation declaration followed by an attribute definition list that sets options that control the architecture engine. In this case, *object* is the top-level document element in the architectural document (ArcDocF), *cellar* is the architectural form attribute (ArcFormA), *cellarNames* is the "attribute renamer" attribute (ArcNamrA; see below for an explanation), and *cellar.dtd* (see Figure 2) is the architectural DTD (ArcDTD). Like the DTD for the original document, the mapping DTD is also for ⟨!DOCTYPE tei.2⟩; thus the original DTD is included in full without modification at the end.

The bulk of the mapping DTD consists of mapping rules expressed as ATTLIST declarations. In addition to the features described above in section 3, the mapping

```
<!DOCTYPE TEI.2 SYSTEM "textcrit.dtd">
<TEI.2>
<text>
<front>
<docTitle>2 Clement, chapter 7</docTitle>
<witlist>
<wit id=A type=Manuscript>Codex Alexandrinus
<bibl>A Greek uncial of the fifth century.  Housed in the British
Museum.  Published in:  The Codex Alexandrinus in reduced photographic
 facsimile, with an introduction by F. G. Kenyon, London 1909.
</bibl></wit>
<wit id=C type=Manuscript>Codex Constantinopolitanus
   <bibl> . . . </bibl></wit>
<wit id=S type=Manuscript>Syriac Version
   <bibl> . . . </bibl></wit>
<wit id=L type=Edition>Lightfoot 1890
<bibl>Lightfoot, J. B.  1890.  The Apostolic Fathers: Clement,
Ignatius, Polycarp (2nd edition).  Part One: Clement, volume 2, pages 210-
261.
Macmillan.  (Reprinted 1989 by Hendrickson Publishers, Peabody, MA)
</bibl></wit>
<wit id=Lb type=Edition>Loeb edition
   <bibl> . . . </bibl></wit>
<wit id=B type=Edition>Bihlmeyer 1970
<bibl> . . . </bibl></wit>
<wit id=W type=Edition>Wengst 1984
   <bibl> . . . </bibl></wit>
</witlist>
</front>
<body>
<div n=7>
<!-- ***************** Verse 1 ******************** -->
<s n=1>
w(/ste
<app><rdg wit='A L Lb B'>ou)=n</rdg>
   <rdg wit='C S W'><omit></rdg></app>
a)delfoi/
<app><rdg wit='A L Lb B'>mou</rdg>
   <rdg wit='C W'><omit></rdg></app>
a)gwnisw/meqa ei)do/tej, o(/ti e)n xersi\n o(
<app><rdg wit='C S L Lb B W'>a)gw\n</rdg>
   <rdg wit='A'>ai)w/n</rdg></app>
kai\ o(/ti ei)j tou\j fqartou\j a)gw=naj kataple/ousin
polloi/, a)ll' ou) pa/ntej stefanou=ntai,
<app><rdg wit='C L Lb B W'>ei) mh\</rdg>
   <rdg wit='A'>oi( mh/</rdg>
   <rdg wit='S'>ei) mh\ mo/non</rdg></app>
oi( polla\ kopia/santej kai\ kalw=j a)gwnisa/menoi.
</s>
<!-- and so forth for remaining verses  -->
</div>
</body></text>
</TEI.2>
```

*Figure 7.* The sample TEI data file.

rules in Figure 10 use these additional features of the CELLAR architecture (see Figure 2):

- The “attribute renamer” (see, for instance, *cellarNames* under ⟨wit⟩) takes a list of paired names. The architectural attribute which is the first member of a pair takes on the value of the client attribute which is the second member.

```
<!-- TextCrit.DTD

     A DTD for encoding a text critical edition. All tags
     are from the Text Encoding Initiative guidelines.
     The content models have been simplified to use only
     the tags needed for the sample text of II Clement.
     The aim is to faithfully represent the TEI scheme of
     markup without having to deal with the huge TEI DTD.

     This DTD reflects the "Parallel segmentation method"
     of encoding—see TEI Guidelines, section 19.2.3.  -->

<!ELEMENT TEI.2     - - ( text )                        >

<!ELEMENT text      - - ( front, body )                 >

<!ELEMENT front     - - ( docTitle, witList )           >

<!ELEMENT docTitle  - - (#PCDATA)                       >

<!ELEMENT witList   - - ( wit+ )                        >

<!ELEMENT wit       - - ( #PCDATA, bibl? )              >
<!ATTLIST wit       id   ID    #REQUIRED
                    type CDATA #REQUIRED                >

<!ELEMENT bibl      - - (#PCDATA)                       >

<!ELEMENT body      - - ( div+ )                        >

<!ELEMENT div       - - ( s+ )                          >
<!ATTLIST div       n  CDATA  #IMPLIED                  >

<!ELEMENT s         - - ( #PCDATA | app )+              >
<!ATTLIST s         n  CDATA  #IMPLIED                  >

<!ELEMENT app       - - ( rdg+ )                        >

<!ELEMENT rdg       - - ( #PCDATA | omit )              >
<!ATTLIST rdg       wit  IDREFS  #REQUIRED              >

<!ELEMENT omit      - O  EMPTY                          >
```

*Figure 8.* The DTD for the sample TEI data file.

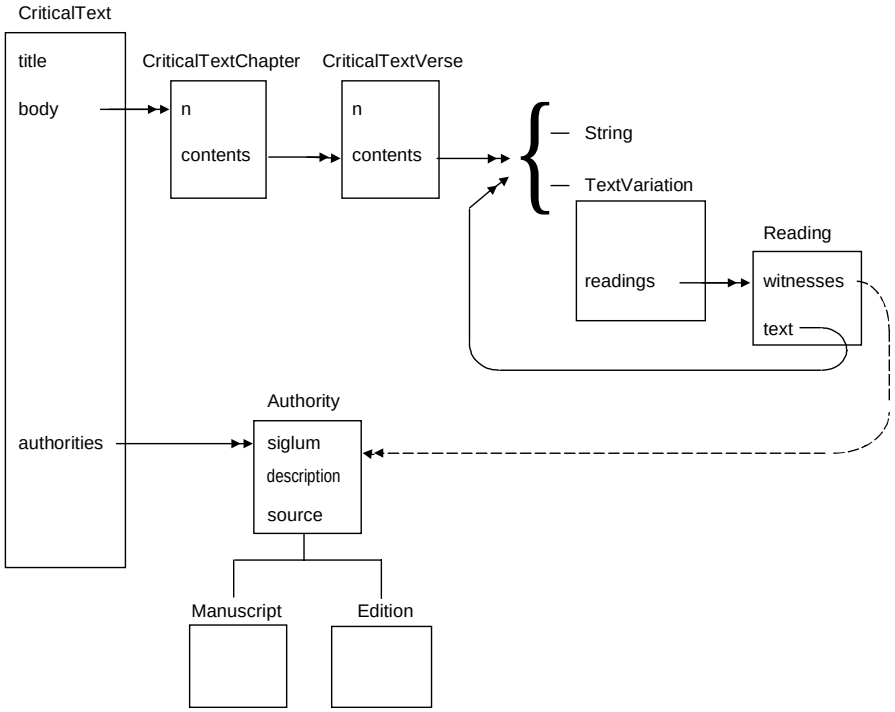


*Figure 9.* The target conceptual model in the object database.

Thus the first pair defined for ⟨wit⟩ says that the name for the *class* of the object to create comes out of the *type* attribute of the client element.

- The three architectural attributes *attrName*, *attrType*, and *attrValue* work together to map an attribute of the client element onto an attribute of the target object. When the *attrType* is IDREFS (as under ⟨rdg⟩), the resulting value is a set of pointers to the objects associated with the given IDs.
- The *encoding* architectural attribute allows one to build a multilingual database (Simons and Thomson, 1998). For instance, the declaration under ⟨s⟩ says that all of the strings in the content of ⟨s⟩ (including all its subelements) should be created with the CELLAR language encoding named "GKOb" (for ancient Greek, beta code).

### 5.3 CREATE A CLIENT DTD THAT INVOKES ARCHITECTURAL PROCESSING

Our sample data file uses a DTD in the file *textcritt.dtd*. We must define an alternate version of this DTD which invokes the desired architectural processing features. The result is given in Figure 11. Note that this DTD does not modify the original declarations for the elements and attributes of the client DTD in any way. Rather, it duplicates them exactly by including the original DTD in full at the end. The purpose of this version of the DTD is to declare that the architecture named

```
<!-- map-textcrit.dtd
     This maps textcrit.dtd onto CELLAR
     architectural forms                    -->

<!afdr "ISO/IEC 10744:1992"
    --Allow multiple ATTLIST declarations--    >

<?ArcBase cellar>
<!ENTITY % cellarDTD SYSTEM "cellar.dtd"       >
<!NOTATION cellar SYSTEM>
<!ATTLIST  #NOTATION cellar
    arcDocF  NAME  #FIXED object
    arcFormA NAME  #FIXED cellar
    arcNamrA NAME  #FIXED cellarNames
    ArcDTD   CDATA #FIXED "%cellarDTD"         >

<!ATTLIST TEI.2
     cellar      NAME  #FIXED object
     class       CDATA #FIXED CriticalText     >

<!ATTLIST text
     cellar      NAME  #FIXED ignore           >

<!ATTLIST front
     cellar      NAME  #FIXED ignore           >

<!ATTLIST docTitle
     cellar      NAME  #FIXED attr
     contentAttr CDATA #FIXED title            >

<!ATTLIST witList
     cellar      NAME  #FIXED attr
     contentAttr CDATA #FIXED authorities      >

<!ATTLIST wit
     cellar      NAME  #FIXED object
     cellarNames CDATA #FIXED "class type
                               attrValue id"
     attrName    CDATA #FIXED siglum
     attrType    CDATA #FIXED String
     contentAttr CDATA #FIXED description
  -- id          automatically preserved from
                 client attr of same name --   >

<!ATTLIST bibl
     cellar      NAME  #FIXED attr
     contentAttr CDATA #FIXED source           >

<!ATTLIST body
     cellar      NAME  #FIXED attr
     contentAttr CDATA #FIXED body             >

<!ATTLIST div
     cellar      NAME  #FIXED object
     class       CDATA #FIXED CriticalTextChapter
     contentAttr CDATA #FIXED contents
     attrName    CDATA #FIXED n
     attrType    CDATA #FIXED String
     cellarNames CDATA #FIXED "attrValue n"    >

<!ATTLIST s
     cellar      NAME  #FIXED object
     class       CDATA #FIXED CriticalTextVerse
     contentAttr CDATA #FIXED contents
     attrName    CDATA #FIXED n
     attrType    CDATA #FIXED String
     cellarNames CDATA #FIXED "attrValue n"
     encoding    CDATA #FIXED GKOb             >

<!ATTLIST app
     cellar      NAME  #FIXED object
     class       CDATA #FIXED TextVariation
     contentAttr CDATA #FIXED readings         >

<!ATTLIST rdg
     cellar      NAME  #FIXED object
     class       CDATA #FIXED Reading
     contentAttr CDATA #FIXED text
     attrName    CDATA #FIXED witnesses
     attrType    CDATA #FIXED IDREFS
     cellarNames CDATA #FIXED "attrValue wit"  >

<!ATTLIST omit
     cellar      NAME  #FIXED object
     class       CDATA #FIXED String           >

<!ENTITY % originalDTD SYSTEM "textcrit.dtd"   >
%originalDTD;
```

*Figure 10.* The meta-DTD for mapping the TEI data into the object architecture.

```
<!-- my-textcrit.dtd
     This is a version of textcrit.dtd that invokes
     the mapping to CELLAR architectural forms. -->

<?ArcBase mapping>

<!ENTITY % mappingDTD SYSTEM "map-textcrit.dtd" >
<!NOTATION mapping SYSTEM>
<!ATTLIST #NOTATION mapping
    ArcDocF  NAME  #FIXED "TEI.2"
    ArcDTD   CDATA #FIXED "%mappingDTD" >

<!ENTITY % originalDTD SYSTEM "textcrit.dtd" >
%originalDTD;
```

*Figure 11.* A client DTD that invokes architectural processing.

*mapping* is to be used. This is done with the ⟨?ArcBase mapping⟩ processing instruction. The architectural support declaration that follows specifies that ⟨TEI.2⟩ is the generic identifier for the document element of the architectural document (ArcDocF), and and that *map-textcrit.dtd* is the file that contains the architectural DTD (ArcDTD). Note that for this invocation of the architectural processor, the architectural document and the architectural DTD are what Figure 6 calls the annotated client document and the mapping DTD respectively.

### 5.4 ASSOCIATE THE CLIENT DOCUMENT WITH THE MODIFIED DTD

Before performing the final step of automatic translation, the client document instance (in Figure 7) must be changed to use the modified DTD defined in the preceding step. That is,

⟨!DOCTYPE TEI.2 SYSTEM “my-textcrit.dtd”⟩
⟨TEI.2⟩
  ⟨!– the content is as in Figure 7 –⟩
⟨/TEI.2⟩

### 5.5 RUN THE ARCHITECTURE ENGINE TO TRANSLATE THE DOCUMENT

The next step is to run the architecture engine to perform the translation of the client document instance into an architectural document instance. The parsers in the SP family (Clark, 1997) are able to do this. For instance, the following command line

sgmlnorm -Amapping -Acellar clement.sgm

translates the client document instance into the corresponding document that uses the object markup system of the CELLAR architecture. A fragment of the output is given in Figure 12; compare this to the original file in Figure 7.

```
<object class="CriticalText">
   <attr contentAttr="title" pcdataClass="String">
      2 Clement, chapter 7</attr>
   <attr contentAttr="authorities">
      <object class="Manuscript" id="A"
         contentAttr="description" pcdataClass="String"
         attrName="siglum" attrType="String" attrValue="A">
         Codex Alexandrinus
         <attr contentAttr="source" pcdataClass="String">
            A Greek uncial of the fifth century ... </attr>
      </object>
      <!-- The other six authorities -->
   </attr>
   <attr contentAttr="body">
      <!-- The CriticalTextChapter and its conetnts -->
   </attr>
</object>
```

*Figure 12.* The sample data translated to the object architecture.

### 5.6 PARSING THE ARCHITECTURAL DOCUMENT INTO CELLAR

The final step in the process is to run a method of the CELLAR system that invokes a data input parser that converts the architectural document instance into the corresponding structure of objects. The input to the CELLAR parser is the ESIS output file of the *nsgmls* parser. At the heart of the implementation is a recursive function of 125 lines that processes one element at a time from the ESIS stream. This function relies on another 125 lines of code in smaller supporting functions. The source code for this parser is listed in full and explained in an electronic working paper (Simons, 1997b).

## 6. Conclusion

The CELLAR architecture that has been implemented is actually richer than what is presented above. It also handles cases where: (1) an element does not correspond to anything in the object model so that it must be discarded along with all its content, (2) an element actually corresponds to two objects (one embedded within the other), and (3) the exact mapping relationship is conditioned by the context. The full architecture and a number of complete examples are available in an electronic working paper (Simons, 1997b). The working paper includes all the files needed to run the critical text example discussed in this paper.

The results to date have been promising. The goal of developing a general solution to the problem of importing SGML data into an existing object database schema has been achieved. Given the fact that the method permits superfluous markup to be ignored and unmappable elements to be discarded altogether, it is always possible to achieve a translation from an SGML file into a structure of objects in the database. However, important information could be lost in the

process. In the final analysis, the usefulness of the result depends on the degree of congruence between the conceptual model of the markup for the source data in SGML and that of the schema for the target object database.

## Acknowledgments

I am deeply indebted to my colleague Robin Cover who has helped in many ways over the course of this project. He has gone the extra mile in helping me to find resources and in offering useful feedback and encouragement.

## References

Booch, G. *Object-Oriented Analysis and Design with Applications*, 2nd ed. Redwood City, CA: Benjamin/Cummings Publishing Co., 1994.

Borgida, A. "Features of Languages for the Development of Information Systems at the Conceptual Level". *IEEE Software*, 2(1) (1985), 63–72.

Cattell, R.G.G. et al. *The Object Database Standard 2.0*. San Francisco: Morgan Kaufman, 1997.

Clark, J. *SP: An SGML System Conforming to International Standard ISO 8879 – Standard Generalized Markup Language*, version 1.2, 1997 ⟨http://jclark.com/sp/⟩. See especially "Architectural Form Processing" ⟨http://jclark.com/sp/archform.htm⟩.

Coad, P. and E. Yourdon. *Object-Oriented Analysis*, 2nd ed. Englewood Cliffs, NJ: Prentice-Hall, Inc, 1991.

Cover, R. "Architectural Forms and SGML/XML Architectures". *The SGML/XML Web Page*, 1998 ⟨http://www.sil.org/sgml/topics.html#archForms⟩.

DeRose, S. and D. Durand. *Making Hypermedia Work: A User's Guide to HyTime*. Boston: Kluwer Academic Publishers, 1994. See especially pages 79–90.

Harbo, K., J. Engelen, F. Evenepoel and B. Bauwens. "Document Processing Based on Architectural Forms with ICADD as an Example". A paper presented at the SGML BeLux '94 Conference, 1994 ⟨http://www.sgmlbelux.be/94/5archfrm.html⟩.

ISO. *Hypermedia/Time-based Structuring Language: HyTime, ISO/IEC 10744*. Geneva: International Organization for Standardization, 1992.

ISO. "Architectural Form Definition Requirements (AFDR)". Annex A.3 of ISO/IEC N1920, *Information Processing – Hypermedia/Time-based Structuring Language (HyTime)*, 2nd ed. 1997-08-01. Geneva: International Organization for Standardization, 1997 ⟨http://www.ornl.gov/sgml/wg8/docs/n1920/html/clause-A.3.html⟩.

Kimber, W. E. "A Tutorial Introduction to SGML Architectures". An ISOGEN International Corporation workpaper, 1997a ⟨http://www.isogen.com/papers/archintro.html⟩.

Kimber, W. E. "An Approach to Literate Programming with SGML Architectures". An ISOGEN International Corporation Workpaper, 1997b ⟨http://www.isogen.com/papers/litprogarch/litprogarch.html⟩.

Kimber, W. E. "Using the RDF Data Model as an SGML Architecture". An ISOGEN International Corporation workpaper, 1997c ⟨http://www.isogen.com/demos/RDF/rdfarch.html⟩.

Megginson, D. "XML Architectural Forms". A posting to the XML-DEV mailing list, 13 December 1997 ⟨http://www.lists.ic.ac.uk/archives/xml-dev/9712/0181.html⟩.

Megginson, D. *Structuring XML Documents.* Charles F. Goldfarb Series on Open Information Management. Upper Saddle River, NJ: Prentice Hall, 1998a.

Megginson, D. "Using the XAF Package for Java". A Megginson Technologies Workpaper, 1998b ⟨http://www.megginson.com/XAF/⟩.

SIL. *CELLAR Web Page*. Summer Institute of Linguistics Web Site, 1998 ⟨http://www.sil.org/cellar⟩.

Simons, G. “Conceptual Modeling Versus Visual Modeling: A Technological Key to Building Consensus”. *Computers and the Humanities*, 30(4) (1997a), 303–319.

Simons, G. “Importing SGML Data into CELLAR by Means of Architectural Forms”. A Summer Institute of Linguistics Workpaper, 1997b ⟨http://www.sil.org/cellar/import/⟩.

Simons, G. “Using Architectural Forms to Map SGML Data into an Object-Oriented Database”. In *Proceedings of SGML/XML ’97, Washington, D.C., 8–11 December 1997.* Graphics Communications Association, 1997c, pp. 449–459.

Simons, G. and J. Thomson. “Multilingual Data Processing in the CELLAR Environment”. In *Linguistic Databases.* Ed. J. Nerbonne, Stanford, CA: Center for the Study of Language and Information, 1998, pp. 203–234. (The original working paper is available at ⟨http://www.sil.org/cellar/mlingdp/mlingdp.html⟩.)

Sperberg-McQueen, C.M. and L. Burnard. *Guidelines for Electronic Text Encoding and Interchange.* Chicago and Oxford: Text Encoding Initiative, 1994.