FieldWorks XML model

Ken Zook

October 20, 2008

Contents

1 XML introduction

XML (Extensible Markup Language) is a standard file format for storing data along with its semantic markup. The official description is found at www.w3.org/TR/REC-xml. It provides numerous advantages over SFM including attributes and clear structural markers. Start with a simple SFM dictionary entry:

```
\lx abal
\ps adj
\gl turbulent
\de Turbulent with rip tide.
\xv Maabal tuud ha tungud Duway Bullud.
\xe The current is very turbulent near Duway Bullud.
\ps v
\gl become turbulent
```

```
\de For the ocean (a rip tide) to become turbulent.
\xv Umabal tuud in dagat bang hunas.
\xe The sea becomes very turbulent at low tide.
\xv Magabal in dagat bang lumabay in kappal.
\xe There will be rip tide when a boat passes.
\dt 18/May/2004
```

This can be turned into a basic XML file:

```
<?xml version="1.0" encoding="UTF-8"?>
<SFMDatabase>
<lx>abal</lx>
<ps>adj</ps>
<gl>turbulent</gl>
<de>Turbulent with rip tide.</de>
<xv>Maabal tuud ha tungud Duway Bullud.</xv>
<xe>The current is very turbulent near Duway Bullud.</xe>
<ps>v</ps>
<gl>become turbulent</gl>
<de>For the ocean (a rip tide) to become turbulent.</de>
<xv>Umabal tuud in dagat bang hunas.</xv>
<xe>The sea becomes very turbulent at low tide.</xe>
<xv>Magabal in dagat bang lumabay in kappal.</xv>
<xe>There will be rip tide when a boat passes.</xe>
<dt>18/May/2004</dt>
</SFMDatabase>
```

The top line is a special format that identifies this as an XML file and identifies the encoding. XML files usually use Unicode UTF-8 data. Following this is a single top-level XML element required by XML. XML elements *must* have a “begin” tag (enclosed in wedges) and a matching “end” tag (enclosed in wedges with a slash after the opening wedge). Inside the element can be embedded elements or raw text. The SFMDatabase element contains a sequence of elements, while the remaining elements contain raw text. This example starts each element on a new line, but that is optional. An XML processor will allow any amount of white space between elements.

Additional information can be provided that makes the data more useful:

```
<?xml version="1.0" encoding="UTF-8"?>
<SFMDatabase>
  <entry>
    <lx lang="tsg">abal</lx>
    <sense>
      <ps lang="en">adj</ps>
      <gl lang="en">turbulent</gl>
      <de lang="en">Turbulent with rip tide.</de>
      <example>
        <xv lang="tsg">Maabal tuud ha tungud Duway Bullud.</xv>
        <xe lang="en">The current is very turbulent near Duway Bullud.</xe>
      </example>
    </sense>
    <sense>
      <ps lang="en">v</ps>
      <gl lang="en">become turbulent</gl>
      <de lang="en">For the ocean (a rip tide) to become turbulent.</de>
      <example>
        <xv lang="tsg">Umabal tuud in dagat bang hunas.</xv>
        <xe lang="en">The sea becomes very turbulent at low tide.</xe>
      </example>
      <example>
        <xv lang="tsg">Magabal in dagat bang lumabay in kappal.</xv>
        <xe lang="en">There will be rip tide when a boat passes.</xe>
      </example>
    </sense>
    <dt value="18/May/2004"/>
  </entry>
</SFMDatabase>
```

Some additional structure has now been added to the data. An Entry element surrounds the entry, a Sense element surrounds senses, and an Example element surrounds examples. Attributes have also been added to identify the language of each string. Attributes are placed inside the begin element after the element name. Elements may have 0 or more attributes. Each attribute has a name and the contents enclosed in double or single quotes. The date element has also been changed to include the date in a value attribute rather than inside the element. If the element does not have any contents, an abbreviated form of the end tag can be used (a slash just before the closing wedge of the begin tag).

Within attribute data and embedded element data, certain characters *must* be escaped to avoid confusion with the markup around the data. The main characters you need to escape in data are as follows:

<	⇨ &lt;
>	⇨ &gt;
&	⇨ &amp;
"	⇨ &quot;

These are predefined entities. For example, to indicate "ISO-8859-1<>UNICODE" the XML data needs to be "ISO-8859-1&lt;&gt;UNICODE". Actually, any Unicode value can be escaped using the hexadecimal numeric character references, such as “&#x12a3;”.

There are various kinds of XML editors that can make editing XML files easier. XMLmind is a free XML editor that can be customized in various ways. Andy Black provides customizations for working with linguistics papers called XLingPap. He also provides a way in Flex to dump interlinear text to XML that works with these customizations. ZEdit handles large XML files with ease, although it does not provide any special XML help.

Microsoft Internet Explorer has a nice ability to check the basic integrity of an XML file. By default, when you double-click an XML file, it opens in Internet Explorer and displays it with buttons to expand and contract element hierarchy. If something invalid is in the file, it usually gives a useful error message near the end at the point where it failed.

You can also validate an XML file further with a DTD or schema. FieldWorks provides FwDatabase.dtd in c:\Program Files\SIL\FieldWorks\Data that can validate a FieldWorks XML file.

Once your data is in XML, you can use processes such as XSLT to transform it in numerous ways. This can work in conjunction with style sheets to display the data in many interesting ways.

Comments in XML are ignored in processing and are enclosed in <!-- and -->:

```
 <!-- this is a comment -->
```

2 FieldWorks XML model

FieldWorks provides a standard way to represent the conceptual model as XML data. The program can dump user data in this FieldWorks XML format and load it back into a database. The LinguaLinks Workshops and SFM import processes transfer the input files into this format. It is then imported into existing FieldWorks Projects. This section describes the FieldWorks XML model.

In addition to user data, the FieldWorks database has many non-data sections such as stored procedures, views, and triggers. None of these are dumped in a FieldWorks XML file. When loading a FieldWorks XML file into a database, FieldWorks starts with a database that already has these sections loaded.

In FieldWorks, class names are unique, but property names are not. Thus, you can have a Name for a CmPossibility or a CmMajorObject. In order for a DTD to validate an XML file, all element names *must* be unique. To do so, property names *always* have the class number appended to the property name. This is the class on which the property is defined, not necessarily the class of the current instance. As a result, a Name property for the LexDb is Name5 because it inherits from CmMajorObject (which has a class id of 5). The Name property for a PartOfSpeech is Name7 because it inherits from CmPossibility (which has a class id of 7). You can use c:\Program Files\SIL\FieldWorks\Helps\ModelDocumentation.chm to find class numbers as described in Conceptual model overview.doc. Each conceptual model class and property is stored in the XML file as elements. If there are no contents, the elements are usually omitted.

2.1 Basic properties

2.1.1 Strings

All strings other than Unicode or BigUnicode are identified with a writing system code (e.g., ws="en"). This writing system is the ICULocale of your writing system. To see this code in a FieldWorks program, go to the Writing System Properties dialog then click the Advanced button. The code is at the bottom of the dialog.

In the XML representation of strings, there is no difference between the short and long versions. As a result, you need to consider four basic kinds of strings.

2.1.1.1 Unicode or BigUnicode

The simplest type is a FieldWorks Unicode or BigUnicode string.

```
<EthnologueCode6001>
<Uni>XKAL</Uni>
</EthnologueCode6001>
```

This is the EthnologueCode property in LangProject (class = 6001). This kind of code and filenames do not have an inherent language, so they are specified as FieldWorks Unicode strings. There is a slight problem with this. FieldWorks needs to use a writing system in order to determine which font to use to display the string. At this point the FieldWorks display mechanism shows boxes for characters that are not in the current font rather than substituting other fonts as needed. FieldWorks generally chooses a writing system representing the system language on your computer, if the writing system exists, otherwise it defaults to English when displaying these strings.

2.1.1.2 String or BigString

Normally a string will have a single run, but to demonstrate some of the complexity you may encounter, this demonstrates how a complex FieldWorks String or BigString is represented:

```
<Contents16>
<Str>
<Run ws="en">Went fishing with </Run>
<Run ws="de">Rolf</Run>
<Run ws="en"> at the </Run>
<Run ws="en" tags="I2121F090-4744-4BC0-AE46-3E0D48C071B5 I2BDD0E85-F9B2-
11D3-977B-00C04F186933">river</Run>
<Run ws="en" >. Within no time, he had caught about 10 </Run>
<Run ws="en" bold="invert" fontFamily="Adams Extended">fierce</Run>
<Run ws="en"> looking </Run>
<Run ws="en" namedStyle="Emphasized Text">pirana</Run>
<Run ws="en"> which we </Run>
<Run ws="en"
externalLink="silfw://localhost/link?app%3dLanguage+Explorer%26tool%3dlexiconEdit%2
6guid%3de55e9139-d219-4359-8e36-
355cfd423e74%26server%3d.%5cSILFW%26database%3dTestLangProj"
namedStyle="External Link">subsequently</Run>
<Run ws="en"> cleaned and ate for </Run>
<Run ws="en" externalLink="C:\Dropbox\Cooked pirana.jpg" namedStyle="External
Link">breakfast</Run>
<Run ws="en">.</Run>
<Run ws="en"
type="picture">424D262C00000000000036…04000028000000860000</Run>
</Str>
</Contents16>
```

This is the Contents property of an StTxtPara (class = 16) object. It has many embeddings to demonstrate the rich capabilities of FieldWorks strings. FieldWorks Strings contain a sequence of Runs inside a Str element, with each run specifying a writing system. All characters in a given run share the same attributes. Whenever a set of attributes changes, it requires a new Run. This FieldWorks String starts with a Run of English characters, then contains an embedded German word (ws="de"). The word "river" has two overlay tags applied to the word. These are used in the data notebook to allow portions of strings to reference list items. The GUIDs in these cases are the FieldWorks IDs for two list items. The word "fierce" is hard-coded by inverting the bold toggle and setting a specific font. The word "pirana" is tagged with a style. The formatting associated with the Emphasized Text style is determined by an StStyle elsewhere in the system. The string "subsequently" is formatted with an External Link style, plus it holds a hot link to a FieldWorks lexical entry. The word "breakfast" also has an External Link style and references an external picture file. The final run represents an embedded object, which in this case is a picture, encoded as a long string of hex codes. FieldWorks does not use this way of inserting pictures anymore, but this demonstrates how other objects can be embedded in the string.

In the database, all of this non-text information is compressed into a binary format field which cannot be analyzed by SQL code. Thus, XML files provide the only way to safely work with FieldWorks strings.

The value for the “ws” attribute in strings is the ICULocale property of LgWritingSystem.

2.1.1.3 MultiUnicode or MultiBigUnicode

Many properties (fields) in FieldWorks are designed to hold strings that will never have embedding, but may have translations in different languages or writing systems (e.g., lexeme form, gloss, and names and abbreviations for list items). Here is how these MultiUnicode or MultiBigUnicode properties are encoded.

```
<Name7>
<AUni ws="en">adjunct</AUni>
<AUni ws="fr">accessoire</AUni>
<AUni ws="es">adjunto</AUni>
</Name7>
```

This field is the Name for a CmPossibilityItem (class = 7) object. In this case, there are three FieldWorks Unicode strings; English, French, and Spanish. Each FieldWorks Unicode string is stored in an AUni element. Even though the FieldWorks Unicode string itself does not have a writing system, FieldWorks knows the intended writing system by the “ws” attribute of the AUni element.

2.1.1.4 MultiString or MultiBigString

Other properties (fields) need the ability to store equivalent translations, but also need to allow things such as embedded writing systems and formatting. Here is how these MultiString or MultiBigString properties are encoded.

```
<Translation29>
<AStr ws="en">
<Run ws="en">You may damage the computer.</Run>
</AStr>
<AStr ws="fr">
<Run ws="fr">Vous risquez d'endommager l'ordinateur.</Run>
</AStr>
</Translation29>
```

This field is the Translation property on a CmTranslation (class = 29) object. This case has translations for English and French. This is a simple FieldWorks String in that it only contains one run, but any number is possible. Each FieldWorks String is stored in an AStr element which identifies the overall writing system for the FieldWorks String. Normally this is the same writing system as the first run, but it does not have to be. You could have an English string that starts with a French word.

2.1.2 Other basic properties

Integer properties are represented as follow:

```
<HomographNumber5002><Integer val="1"/></HomographNumber5002>
```

This is the HomographNumber property on LexEntry (class = 5002). The value of the integer is stored as a decimal "val" attribute. FieldWorks normally suppresses the property altogether if the value is zero, since that is the default.

Boolean properties are represented as follows:

```
<IsSorted8><Boolean val="true"/></IsSorted8>
```

This is the IsSorted property on CmPossibilityList (class = 8). The value is stored in the "val" attribute as "true" or "false". FieldWorks normally suppresses the property altogether if the value is false, since that is the default.

GUID properties are represented as follows:

```
<StylesheetVersion3001>
<Guid val="631E38DC-9E83-4078-A405-EA78D79CB7E8"/>
</StylesheetVersion3001>
```

This is the StylesheetVersion property on Scripture (class = 3001). The GUID value is stored in the "val" attribute in this string format.

Binary properties are represented as follows:

```
<Details20><Binary>6400000001000000</Binary></Details20>
```

This is the Details property on UserViewField (class = 20). Binary data is stored as a sequence of hexadecimal bytes.

Time properties are represented as follows:

```
<DateModified5><Time val="2005-06-06 15:46:32.710"/></DateModified5>
```

This is the DateModified property on CmMajorObject (class = 5). The time is stored in the "val" attribute in the format YYYY-MM-DD HH:MM:SS.TTT.

GenDate properties are represented as follows:

```
<DateOfEvent4006><GenDate val="200109131"/></DateOfEvent4006>
```

This is the DateOfEvent property on RnEvent (class = 4006). The generic date is stored as a decimal number in the "val" attribute.

2.2 Class instances

Instances of classes are stored in an XML file as an element with the class name. Here is an example of the back translation CmPossibility item:

```
<CmPossibility id="I80A0DDDB-8B4B-4454-B872-88ADEC6F2ABA">
<DateCreated7><Time val="2004-09-10 21:17:31.153"/></DateCreated7>
<DateModified7><Time val="2004-09-10 21:19:00.000"/></DateModified7>
<Name7>
<AUni ws="en">Back translation</AUni>
</Name7>
<Abbreviation7>
<AUni ws="en">BT</AUni>
</Abbreviation7>
</CmPossibility>
```

The properties on the class are stored as nested elements. Classes have optional “id” attributes that hold a GUID with “I” attached to the front. XML “ids” cannot start with a digit, so the “I” prevents this from happening. An “id” attribute is only needed if the instance is being referenced by some other object in the XML file. The “id” can actually be almost any alphanumeric string. If it is the above GUID format, that exact GUID is stored in the database. If it is anything else, a new GUID is generated for the object when loaded into the database.

2.3 Owning properties

In the XML file there is no distinction between atomic, sequence, and collection owning properties. The instance(s) are simply stored as nested elements in the contents of the owning property element.

```
<Senses5002>
<LexSense id="IC3569FFB-967B-4191-BEAA-D92EB3A45166">
<Gloss5016>
<AUni ws="en">to see</AUni>
</Gloss5016>
</LexSense>
<LexSense id="IA3569FFB-967B-5191-EEAA-D92EB3A45164">
<Gloss5016>
<AUni ws="en">to understand</AUni>
</Gloss5016>
</LexSense>
</Senses5002>
```

This example illustrates two LexSense objects owned in the Senses property of LexEntry (class = 5002). For sequence properties, the order of the owned objects is important. The order is irrelevant for collections. The load process does not catch multiple objects in an atomic owning property. It just ignores one. (This needs to be verified, but I think both actually get stored and it isn’t until we dump the imported data back to XML that the log file flags an error.)

2.4 Reference properties

In the XML file there is no distinction between atomic, sequence, and collection reference properties. The reference(s) are simply stored as nested Link elements in the contents of the reference property element. There are various optional attributes for a Link element. Whenever FieldWorks dumps the database, it uses actual GUIDs in all targets, plus it attempts to fill in other attributes to aid the human reader. On import, the additional attributes provide alternate ways to find the specified target if the target attribute is missing.

```
<Confidence4004>
<Link target="I2BDD1100-F9B2-11D3-977B-00C04F186933" ws="en" abbr="hi"
name="High"/>
</Confidence4004>
```

This is the Confidence property of RnEvent (class = 4004). It references a CmPossibility. The Link element needs enough information to identify the target object. As long as the target attribute is present, it is used when loading the data. The target attribute in this case must match the “id” attribute of the target CmPossibility. As described earlier, this may or may not be a real GUID. When constructing “ids” by hand, it is usually easier using something else. The important thing is that all “ids” in the XML file must be unique.

```
<Confidence4004><Link target="A234"/></Confidence4004>
….
<CmPossibility id=”A234”>
</CmPossibility>
```

This will work fine to link Confidence to the CmPossibility. When it is imported, a new GUID is generated for the CmPossibility. This assumes you are starting with a blank database. If you are trying to import into an existing database that already has confidence level items and you want to match up to an existing item, you cannot do it this way. There are several ways to prepare a reference to do this:

- <Link target="I2BDD1100-F9B2-11D3-977B-00C04F186933" ws="en" abbr="hi" name="High"/>
- <Link target="I2BDD1100-F9B2-11D3-977B-00C04F186933"/>
- <Link ws="en" abbr="hi" name="High"/>
- <Link ws="en" abbr="hi"/>
- <Link ws="en" name="High"/>

Any of these Links will work, assuming FieldWorks already has the target in the database. In the first two cases, the import process uses the target and ignores the other attributes. When the target attribute is missing and a name is given with a writing system, the import tries to find an item in the appropriate list that matches the name in that writing system. Otherwise if the “abbr” attribute is given with a writing system, the import tries to find an item in the appropriate list that matches the abbreviation in that writing system. These alternate approaches are useful when importing SFM or LinguaLinks Workshops data into an existing database that already has lists. If the desired target is not found, the import process will create an item in the root of the list and set a reference to the new item. When name and “abbr” are included, the name and abbreviation for the new item are set to these values. If only one is given, the name and abbreviation are both set to whatever is given.

Caution: When you are importing an entire database using the db load option, the target attribute is normally essential since the names and abbreviations may not be present in memory when a link is processed. The targets are always known because they are read during the first pass of reading the file.

```
<Targets5120>
<Link wsv="xkal" entry="pub2"/>
<Link wsv="xkal" sense="pala 1.2"/>
<Link wsv="xkal" sense="pus1 2"/>
</Targets5120>
```

This is the Targets property on LexReference (class = 5120). This property, as well as MainEntriesOrSenses on LexEntry can reference both senses and entries. If a target attribute is missing for the link, the import attempts to find the entry or sense specified and sets a link to that. If it cannot find the target entry or sense, it adds a warning message to ImportResidue to identify the problem. In these Link elements, the first one looks for an entry with a headword of 'pub' in the "xkal" writing system and a homograph number of 2. The headword is the citation form if it exists, otherwise the lexeme form. The second links to the second subsense of the first sense of the entry with a headword of 'pala'. The third links to the second sense of the entry with headword 'pus' and homograph number 1. If the sense number is omitted for a sense, it assumes the first sense.

```
<LexicalRelations5016>
<Link wsa="en" abbr="syn" wsv="xkal" sense="kala"/>
</LexicalRelations5016>
```

This is a fake property on LexSense that is only used for import. The actual links are made from LexReference objects, but this format simplifies specifying lexical relations. When this is imported, Flex

- finds a LexRefType with an English abbreviation of "syn"
- creates or uses an existing LexReference item, and
- sets a reference from this to the first LexSense owned on the entry with a headword of 'kala' in the "xkal" writing system as well as a reference from the current sense.

```
<ReversalEntries5016>
<Link ws="en" form="house"/>
</ReversalEntries5016>
```

This is the ReveralEntries property on LexSense (class = 5016). On import, this tries to find a ReversalIndexEntry with an English ReversalForm of "house" and sets a link to that entry. If it cannot find one, it creates a new ReversalIndexEntry in the English ReversalIndex, sets the ReversalForm to "house", and sets a link to that.

```
<SemanticDomains5016>
<Link abbr="2.6.6.3" ws="en" name="Doctor, nurse" wsv="xkal" namev="Omusanu,
omujanjabi"/>
</SemanticDomains5016>
```

This is the SemanticDomains property on LexSense (class = 5016)—a special form only used for import that allows a vernacular form to be set on a semantic domain. This looks for a

CmSemanticDomain with an English abbreviation of 2.6.6.3 and an English name of “doctor, nurse” and sets a link to that domain if it exists. If it does not, it creates one in the root of the semantic domain list. In either case, it sets the name for the “xkal” writing system to “Omusanu, omujanjabi”.

2.5 Top level XML

This illustrates the top level of a FieldWorks XML file:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE FwDatabase SYSTEM "FwDatabase.dtd">
<FwDatabase version="200146">
….
</FwDatabase>
```

The top element is FwDatabase which has an optional version attribute that gives the version of the database. Other than issuing a warning in the import log file, the version number is not used. After import, the database version is set to the version expected by the FieldWorks program.

There are several possible elements in the FwDatabase contents. None of these elements have owners. These are the possibilities:

- A single AdditionalFields element defining custom fields
- A single LangProject element holds the actual project data
- All LgWritingSystem elements defined for this project
- A single ScrRefSystem element
- All UserView elements
- Any custom CmPossibilityList elements created by the user

2.6 Custom fields

When custom fields are defined, an AdditionalFields element is added above the LangProject that defines the custom fields.

```
<AdditionalFields>
<CustomField name="custom" flid="5002500" class="LexEntry" type="String" big="0"
wsSelector="-1" userLabel="My single-string field"/>
<CustomField name="custom1" flid="5002501" class="LexEntry" type="MultiUnicode"
big="0" wsSelector="-3" userLabel="My multi-string field"/>
</AdditionalFields>
```

This specifies the custom field properties that are stored in the Field$ table in the database. The field number for custom fields start at 500. The first custom field has a name of “custom”. The next one is “custom1”, and the number continues to increment for additional ones. At this point the only type of custom fields that are supported in Flex are String and MultiUnicode, and they can only be added to LexEntry and LexSense. In the Data Notebook, several additional types are available. The userLabel is the name displayed in the UI rather than the built-in custom names.

```
<Custom name="custom">
<Str>
<Run ws="en">Some data in my single-string field</Run>
</Str>
```

```
</Custom>
…
<CustomStr name="custom1">
<AUni ws="en">Some data in my multi-string field</AUni>
</CustomStr>
```

This illustrates data stored in the two types of custom fields supported in Flex. The single string field uses a Custom element and the multistring field uses a CustomStr element. Both elements have a name attribute that stores the built-in name for the field.

Note at this point that the types in the Custom Fields dialog starting with "All" are MultiUnicode, so cannot have any embedding.

A FieldWorks SFM import can import data into pre-existing custom fields using the approach described here. Custom Fields can be created in the import dialog, but the Phase1Output.xml file that is actually imported does not contain the AdditionalFields element at this point, so it only works with a database that has appropriate custom fields predefined.

3 Source of FieldWorks XML files

Complete FieldWorks XML files can be produced in two ways:

- In the Backup and Restore dialog, check "Include a human readable (XML) backup". The resulting XML file is zipped into the FieldWorks backup file.
- Use the db program with a "dump" command.

Partial FieldWorks XML files are produced by LinguaLinks Workshops or SFM import as the final step prior to importing. See Technical Notes on LinguaLinks Database Import.doc or Technical Notes on SFM Database Import.doc for more information. These documents describe special techniques that allow changes to be made to these files prior to the actual import, when useful.

C:\Program Files\SIL\FieldWorks\Templates has a number of XML files that use this format:

- NewLangProj.xml
- POS.xml
- SemDom.xml
- OCM-Frame.xml
- OCM.xml

4 Modifying data in FieldWorks XML files

There are times when you may need to make some consistent changes to your data that are not possible in bulk edit. If there are only a few places, it is obviously best to do it manually. However, if hundreds or thousands of consistent changes are needed, and you have the ability to use CC, XSLT, Python, or some other program that can deal with XML data, it may well be worth the time to make the changes in the XML file.

Using SQL on the database is limited in that it cannot deal with strings that have embedded styles or writing systems. The only safe way to make these changes is using IronPython or in a FieldWorks XML file where this information is all clearly available.

Most examples here use CC tables. If you don't have SIL Consistent Changes program installed, it can be downloaded from www.sil.org/computing/catalog/index.asp.

Note: In order to support the possibility of using the Firebird database engine in addition to Micrsoft SQL Server, and due to limited length of names in Firebird, some class and property names were shortened in FieldWorks 5.4 compared to earlier versions. The spreadsheet, Model name changes.xls, lists the changes that were made. If you had cc tables, xls files, etc. for older versions you may need to make a few of these name changes for it to continue to work in FieldWorks 5.4 and later.

Caution: As with any method for modifying the database outside of a FieldWorks program, if you do not know what you are doing, you can inadvertently damage the data. FieldWorks applications may no longer run or it could do damage in a way that will not show up until later. Be *extremely* cautious about making *any* changes to the XML file. Any time you plan to do this, make sure you *first* back up your project and then after the changes are made, check your changes carefully before going on to other work. If the validation process does not show any errors, and the load log file does not show any problems, you are fairly safe, but it is still possible to do things that will cause crashes.

Here are the basic steps for making changes in this way:

1. Use “db dump” command to dump your project to an XML file.
2. Make whatever changes are necessary to the XML file.
3. Use “db validate” command to validate that your changes have not damaged the structure.
4. Use “db load” command to load the changed data into your project.
5. Check the import log file to make sure there were no other problems detected.
6. Check your data in FieldWorks programs to make sure the desired things happened.

The sections below give several examples to illustrate some of the techniques for using the XML file to do something that is not possible to do within the current programs. These examples illustrate how CC can be used for many tasks, but it imposes severe limitations on some types of operations, especially following links. XSLT, Perl, and Python are probably all better choices to use if you are familiar with those languages, or you need to do a lot of work in XML files.

4.1 Making spelling/orthographic changes in one writing system

Suppose you need to make some spelling or orthographic changes, e.g., to all the Tausug strings in the database, whether they are embedded or not. With bulk edit, you can do this on a field-by-field basis (although not when embedding is involved). There may be many other strings in the database (such as Scripture, data notebook, and lists) that use that same writing system that should also be changed. At this point there is no way to do this inside any FieldWorks program. It is basically impossible to do this with a SQL script or IronPython program because of embedded writing systems in strings. Even without that, tracking down all the strings in a given writing system is rather complex in any other approach. However, this task is quite simple when working with a FieldWorks XML file.

Outside of strings in Links attributes, every other string for a given writing system is in *one* of these two environments:

- <Run ws="tsg">a string in Tausug</Run>
- <AUni ws="tsg">a string in Tausug</AUni>

There could be additional attributes in the Run element, but each Run or AUni that has the “tsg” writing system will contain Tausug text. If you just dumped from a database and plan to import the results, these are the only environments you need to change.

If you want to be complete, there may be other Tausug strings in Link attributes. These are ignored when importing from an XML file just dumped from FieldWorks since all dumped Link elements have target values, so the rest of the attributes are ignored. However, if you modify an XML file in preparation for import, you should change these attribute strings as well. There is a limited set of attributes that can occur in a Link element, and current versions of these files will always have writing systems defined prior to the attributes that need them. Here is the list:

writing system defined in ws or wsa attributes ⇨ affects abbr, name, and form attributes
writing system defined in wsv attribute ⇨ affects entry, sense, and namev attributes

This CC table will make this change:

```
c  Change in writing system.cc
c  To set up, set the ws in the begin statement and put your changes in changes group

begin > store(ws) 'tsg' endstore use(main)

group(changes)
'f' > 'v'
'F' > 'V'
'c' > 'k'
'C' > 'K'

group(main)
'<Run ' > next
'<AUni ' > dup use(run)
'<Link ' > dup use(link)

group(run)
'ws="' cont(ws) '"' > dup set(change)
'>' > dup if(change) use(changes,inRun) else use(main) endif

group(inRun)
'</AUni>' > next
'</Run>' > dup clear(change) use(main)

group(link)
'wsv="' cont(ws) '"' > dup set(wv)
'wsa="' cont(ws) '"' > next
'ws="' cont(ws) '"' > dup set(wa)
'abbr="' > next
'form="' > next
'name="' > dup if(wa) use(changes,inLink) endif
'entry="' > next
'namev="' > next
'sense="' > dup if(wv) use(changes,inLink) endif
'>' > dup clear(wa,wv) use(main)
```

```
group(inLink)
"" > dup use(link)
```

4.2 Removing a writing system

You may have some old writing systems that you need to get rid of. FieldWorks 4.0 does *not* have a way to remove these from the database. The Hide button in the FieldWorks Project Properties Writing System tab only removes the writing system from the reference properties stored in LangProject. The actual writing systems are still in the database, and the next time you open this database, it creates corresponding language definition files and ICU changes for these “hidden” writing systems. InstallLanguage.exe can remove the language definition files and changes in ICU, but it does not touch databases. At this point, the only way to get rid of these writing systems is via XML, SQL, or some other program that works directly on the database. If you have text in the writing system you want to delete, the *only* safe way to remove the writing system is to use XML.

To remove a writing system from an XML file, you need to identify two pieces of information for the writing system you want to remove. To find these, open the XML file in ZEdit and search for <LgWritingSystem and search until you find the one you want to remove. You can tell from the Name24 or the ICULocale24 attributes. After you find it, record the GUID in the “id” attribute of LgWritingSystem and record the ICULocale24 contents inside the Uni element. This is the writing system code you can see in the Writing System Properties dialog when you click Advanced. What you do at this point depends on what you want to do with any text that happens to use that writing system: switch everything in an older writing system to a newer one, or just delete everything related to a given writing system. The program can do it automatically, or you can do it manually with ZEdit. The goal is

- to remove the LgWritingSystem element for that writing system, plus a ReversalIndex element if one uses that GUID in WritingSystem5052
- to remove it from the LangProject reference properties, and
- to do something with any strings that are using the writing system.

There are a few other rare places that might use the writing system, such as SortSpec. You should also delete any <AStr elements that use this writing system. When you are done, the GUID and the writing system code should not occur anywhere in the XML file. When searching for the code you can look for the code with double quotes on each side (e.g., "de").

The following CC table will remove a writing system with any strings that use it:

```
c  Remove writing system.cc
c  To set up, set the ws in the begin statement to the ICULocale for the writing system
c  and id to the id for the writing system

begin > store(ws) 'de'
    store(id) 'IA639F4E0-EEF0-11D3-9770-00C04F186933' endstore

group(main)
'<AStr ws="' cont(ws) '"' > use(deleteAstr)
'<Run ' > next
'<AUni ' > store(run) dup use(checkRun)
```

```
'<Link ' > dup use(link)
'<LgWritingSystem id="' cont(id) > use(deleteWritingSystem)
'<AnalysisWss6001' > next
'<CurAnalysisWss6001' > next
'<VernWss6001' > next
'<CurVernWss6001' > dup use(checkWsLink)
'<ReversalIndex ' > store(rev) dup use(inRev)

group(deleteAstr)
'</AStr>' nl > use(main)
'' > omit

group(deleteWritingSystem)
'</LgWritingSystem>' nl > use(main)
'' > omit

group(checkWsLink)
'<Link target="' cont(id) > use(deleteWsLink)
'</' > dup use(main)

group(deleteWsLink)
'/>' nl > use(checkWsLink)
'' > omit

group(inRev)
'<Link target="' cont(id) > endstore use(deleteRev) c Delete this reversal index
'</ReversalIndex>' > out(rev) dup use(main) c Don't change other reversal indexes

group(deleteRev)
'</ReversalIndex>' nl > use(main)
'' > omit

group(checkRun)
'ws="' cont(ws) '"' > endstore use(deleteRun) c Delete these runs
'>' > out(run) dup use(main) c Copy other runs unchanged

group(deleteRun)
'</AUni>' nl > next
'</AUni>' > next
'</Run>' nl > next
'</Run>' > use(main)
'' > omit

group(link)
'wsv="' cont(ws) '"' > set(wv) c Delete this and corresponding attributes
'wsa="' cont(ws) '"' > next
```

```
'ws="' cont(ws) '"' > set(wa) c Delete this and corresponding attributes
'abbr="' > next
'form="' > next
'name="' > if(wa) use(deleteAttr) else dup endif
'entry="' > next
'namev="' > next
'sense="' > if(wv) use(deleteAttr) else dup endif
'>' > dup clear(wa,wv) use(main)

group(deleteAttr)
'"' > use(link)
'' > omit
```

Note: This table will leave any ReversalEntries5016 links that refer to items in the deleted ReversalIndex. This will show up in the validate process and will show up in the error log for import. The import will do the correct thing to remove these links, so these errors can be ignored.

Here is a CC table that will convert everything that uses one writing system to use another:

```
c  Merge writing system.cc

c  To set up, set wsSrc and wsDst in the begin statement to the source and destination
c  ICULocale for the writing system. Set idSrc and idDst to the source and destination
c  id for the writing system. This has limitations discussed in the documentation.

begin > store(wsSrc) 'de'
    store(idSrc) 'IA639F4E0-EEF0-11D3-9770-00C04F186933'
    store(wsDst) 'en'
    store(idDst) 'IBB99B9AD-E657-11D3-9764-00C04F186933' endstore

group(main)
'ws="' cont(wsSrc) '"' > 'ws="' out(wsDst) '"'
'wsa="' cont(wsSrc) '"' > 'wsa="' out(wsDst) '"'
'wsv="' cont(wsSrc) '"' > 'wsv="' out(wsDst) '"'
'<LgWritingSystem id="' cont(idSrc) > use(deleteWritingSystem)
cont(idSrc) > out(idDst)

group(deleteWritingSystem)
'</LgWritingSystem>' nl > use(main)
'' > omit
```

This table is not robust enough to handle all cases. In particular, if there are strings for both source and destination writing systems in an AStr or AUni, you need to merge both the Runs in the two AStr elements *and also* the duplicate AUni elements . If not, the duplicates are dropped silently on the import process. Neither the validation nor the import log catches this problem. If there are ReversalIndexes for both writing systems, you will get two reversal indexes with the *same* writing system. It is easiest to merge these via ZEdit in the xml file prior to import. Also, if both writing systems are active at the time, you will not get duplicates in the writing system lists in FieldWorks Project Properties. You can correct this by simply removing one in the list. Trying to handle all these cases is possible in one or more CC tables, but is beyond the scope of this introduction.

In addition to clearing a writing system from all of your databases, you should remove the old writing systems from language definition files and ICU as well. See InstallLanguage in Icu and writing systems.doc.

4.3 Creating an example sentence from an ImportResidue field

You may have an SFM marker in LexSense_ImportResidue that you later need to make an example sentence. This CC table will extract the \xy field from LexSense_ImportResidue and create an example for each \xy field, even if it occurs multiple times in ImportResidue:

```
c  Move xy to example.cc
c  This extracts \xy fields from LexSense_Import and creates a LexExample for each one.
c  This table doesn't handle writing system or style embedding in the \xy field.

begin > store(sfm) 'xy' c The SFM marker to move
    store(ws) 'xkal' endstore c The vernacular writing system

define(storeExample) > ifneq(ex) ''
    begin
            '<LexExampleSentence>' nl
            '<Example5004>' nl
            '<AStr ws="' outs(ws) '">' nl
            '<Run ws="' outs(ws) '">' outs(ex)  '</Run>' nl
            '</AStr>' nl
            '</Example5004>' nl
            '</LexExampleSentence>' nl
            store(ex) endstore
    end

group(main)
'<ImportResidue5016>' > dup use(ImportResidue5016)
'</Examples5016>' > ifneq(examples) ''
    begin
            out(examples) store(examples) endstore
    end
    dup
'</LexSense>' > ifneq(examples) ''
    begin
            '<Examples5016>' nl
            out(examples) store(examples) endstore
            '</Examples5016>' nl
    end
    dup

group(ImportResidue5016)
'<Run ' > dup use(Run)
```

```
'</ImportResidue5016>' > dup use(main)

group(Run)
'>' > dup use(inRun)

group(inRun)
' \' cont(sfm) ' ' > next
'\' cont(sfm) ' ' > store(ex) use(storingEx)
'</Run>' > dup use(ImportResidue5016)

group(storingEx)
' \' > '\' back
'\' > append(examples) do(storeExample) endstore dup back use(inRun)
'</Run>' > append(examples) do(storeExample) endstore dup use(ImportResidue5016)
```

This is possible because ImportResidue happens to come out near the top of each sense. If this condition ever changes, the CC table will get more complex. Notice that when adding examples, you may need to deal with situations where there are already some examples in the sense as well as where there are none.

4.4 Creating a sub-MDF dictionary

This CC table demonstrates one way to extract dictionary entries from a FieldWorks XML file and output them as MDF codes. It only supports a very limited number of codes but demonstrates techniques that can be used in CC. The order of fields in an XML record is not necessarily the order you want in an MDF file. This table stores the relevant parts until it reaches the end of the example, sense, or entry, then outputs the stored pieces in the correct order:

```
c  Extract sub-MDF dictionary
c  This extracts lexeme form with affix marks, homograph number,
c  part of speech, gloss, definition, example sentences

begin > store(vern) 'xkal' c Vernacular writing system
    store(anal) 'en' endstore c Analysis writing system

define(writeEntry) >
    ifneq(lx) '' begin '\lx ' out(pre,lx,post) end else begin '\lx ??' end nl
    ifneq(hm) '' begin '\hm ' out(hm) nl end
    ifneq(senses) '' begin out(senses) end
    store(lx,pre,post,hm,senses) endstore nl

define(storeExample) > append(examples)
    ifneq(xv) '' begin '\xv ' outs(xv) nl end
    ifneq(xe) '' begin '\xe ' outs(xe) nl end
    store(xv,xe) endstore

define(storeSense) > append(senses)
    '\sn' nl
```

```
        ifneq(ge) '' begin '\ge ' outs(ge) nl end
        ifneq(de) '' begin '\de ' outs(de) nl end
        ifneq(examples) '' begin outs(examples) end
        store(ge,de,examples) endstore

    group(main)
    '</LexEntry>' > do(writeEntry)
    '</LexSense>' > do(storeSense)
    '<LexemeForm5002' > use(lexeme)
    '<HomographNumber5002' > use(homograph)
    '<Gloss5016' > use(gloss)
    '<Definition5016' > use(definition)
    '<LexExampleSentence' > use(example)
    endfile > endfile
    '' > omit

    group(lexeme)
    '<AUni ws="' cont(vern) '">' > store(lx) use(copy,lexeme)
    '<MorphType5035' > endstore use(morphType)
    '</LexemeForm5002' > endstore use(main)
    '' > omit

    group(morphType)
    c  These are fixed guids that never change, although the affix markers could.
    'ID7F713DB-E8CF-11D3-9764-00C04F186933' > store(post) '-' endstore c prefix
    'ID7F713DD-E8CF-11D3-9764-00C04F186933' > store(pre) '-' endstore c suffix
    'ID7F713DA-E8CF-11D3-9764-00C04F186933' > store(pre) '-' store(post) '-' endstore c infix
    '</MorphType5035' > use(lexeme)
    '' > omit

    group(homograph)
    'val="' > store(hm)
    '"/>' > endstore
    '</HomographNumber5002' > use(main)
    '' > omit

    group(gloss)
    '<AUni ws="' cont(anal) '">' > store(ge) use(copy,gloss)
    '</Gloss5016' > endstore use(main)
    '' > omit

    group(definition)
    '<AStr ws="' cont(anal) '">' > store(ret) 'def' store(de) use(astr)
    '</Definition5016' > use(main)
    '' > omit
```

```
group(example)
'</LexExampleSentence' > do(storeExample) use(main)
'<Example5004' > use(vernExamp)
'<Translation29>' > use(analExamp)
'' > omit

group(vernExamp)
'<AStr ws="' cont(vern) '">' > store(ret) 'ver' store(xv) use(astr)
'</Example5004>' > use(example)
'' > omit

group(analExamp)
'<AStr ws="' cont(anal) '">' > store(ret) 'ana' store(xe) use(astr)
'</Translation29>' > use(example)
'' > omit

group(astr)
'<Run' > use(run) c append all runs
'</AStr>' > ifeq(ret) 'def' begin store(ret) endstore use(definition) end
    ifeq(ret) 'ver' begin store(ret) endstore use(vernExamp) end
    ifeq(ret) 'ana' begin store(ret) endstore use(analExamp) end
'' > omit

group(run)
'>' > use(copy,run)
'</Run>' > use(astr)
'' > omit

group(copy)
'</' > dup back(2) excl(copy)
'' > fwd
```

Part of speech is omitted in this CC table because it becomes too hard to calculate. From the sense, you would have to follow the MorphoSyntaxAnalysis5016 link and then depending on which subclass of MoMorphSynAnalysis is instantiated, you would follow the property to the part of speech. To do this in CC would take at least two passes. The first one would construct another CC table that you can then edit to add the remaining changes for the second pass. This is where a more powerful language, such as XSLT, Perl, or Python is much better, although it is still not trivial due to the complexity of the model. Similar complexities exist for all reference properties.

MdfFromFwXml.xsl is an XSLT transform that illustrates how this job can be accomplished using XSLT. This transform actually does considerably more than the CC table above. It not only extracts a \ps field, but nicely handles the various options for stems and affixes. It also puts out sense numbers following \sn and puts out all writing systems for gloss, definitions, and example translations instead of just English. It does leave a lot of blank lines in the output, but these can be cleaned up quickly with CC or search and replace in ZEdit.

To run this transform, you can use the following command at a command prompt in the c:\Program Files\SIL\FieldWorks directory, assuming the transform and XML input file are both in this directory.

```
msxsl TestLangProj.xml MdfFromFwXml.xsl -o mdf.db
```

4.5 Clearing out dictionary, wordform inventory, and text analysis

Suppose you have worked with Flex for some time, but decide that you did a number of things wrong and want to clean things up and start over. You can create a new Fieldworks project, but if you have Scripture and cultural notes in the database, you do not want to lose that data. You cannot do this inside the program, but it is fairly simple to do in a FieldWorks XML file.

The goal is to

- delete the Entries5005 element with all your lexical entries
- delete the Wordforms5063 element with all your wordforms
- delete any TestSets5040 elements from imported wordlists
- clear any Evaluations23 elements that you or the parser created
- clear AdhocCoProhibitions5040 and CompoundRules5040 elements from your morphology
- clear all Members5119 elements holding lexical references
- clear all CmBaseAnnotations and CmIndirectAnnotations used by texts, and
- if you want to clear out your interlinear texts (only the baselines are left at this point), delete the Texts6001 element.

Here is a CC table that will do this:

```
c  Clear dictionary, wordform, and text annotations.cc

group(main)
'<Entries5005>' > next
'<Wordforms5063>' > next
'<Evaluations23>' > next
'<AdhocCoProhibitions5040>' > next
'<CompoundRules5040>' > next
'<TestSets5040>' > next
c  '<Texts6001>' > next c Include this line to delete interlinear texts
'<Members5119>' > use(deleteElement)
'<CmBaseAnnotation' > store(ann) dup use(checkBaseAnnotation)
'<CmIndirectAnnotation' > store(ann) dup use(checkIndirectAnnotation)

group(checkBaseAnnotation)
'target="IEB92E50F-BA96-4D1D-B632-057B5C274132"' > set(del) c wfic
'target="ICFECB1FE-037A-452D-A35B-59E06D15F4DF"' > set(del) c pic
'target="9AC9637A-56B9-4F05-A0E1-4243FBFB57DB"' > set(del) c FT
'target="B63F0702-32F7-4ABB-B005-C1D2265636AD"' > set(del) c TxtSeg
'target="20CF6C1C-9389-4380-91F5-DFA057003D51"' > set(del) c pt
'</CmBaseAnnotation>' nl > endstore
    ifn(del) begin out(ann) dup end clear(del) use(main)

group(checkIndirectAnnotation)
```

```
'target="7FFC4EAB-856A-43CC-BC11-0DB55738C15B"' > set(del) c Nt
'target="B0B1BB21-724D-470A-BE94-3D9A436008B8"' > set(del) c LT
'target="B0B1BB21-724D-470A-BE94-3D9A436008B8"' > set(del) c LT
'</CmIndirectAnnotation>' nl > endstore
    ifn(del) begin out(ann) dup end clear(del) use(main)

group(deleteElement)
'</Entries5005>' nl > next
'</Wordforms5063>' nl > next
'</Evaluations23>' nl > next
'</AdhocCoProhibitions5040>' nl > next
'</CompoundRules5040>' nl > next
'</TestSets5040>' nl > next
'</Texts6001>' nl > next
'</Members5119>' nl > use(main)
'' > omit
```

4.6 Clearing out scripture

Suppose you would like to get rid of everything related to Scripture so that you can start over again. The following CC table will do this.

```
c Remove Scripture from Language Project

group(main)
'<TranslatedScripture6001>' > use(deleteTranslatedScripture6001)
'<UserView ' > store(UserView) dup clear(deleteUserView) use(checkUserView)

c Delete any UserView with a scripture GUID.
group(checkUserView)
'<Guid val="A7D421E1-1DD3-11D5-B720-0010A4B54856"/>' > set(deleteUserView)
'</UserView>' nl > dup endstore ifn(deleteUserView) out(UserView) endif use(main)

c Delete all Scripture info.
group(deleteTranslatedScripture6001)
'</TranslatedScripture6001>' nl > use(main)
'' > omit
```

4.7 Converting bar codes to formatted text

With current versions of Flex you can convert bar codes such as |bxyz|r to emphasized text or some other formatting while importing from SFM. However, this conversion isn't available when importing from LinguaLinks, and it wasn't available in earlier versions of SFM import. If you have these kinds of in-line markers in FieldWorks and would like to convert these to something meaningful, you can use a CC table such as the following one to make these conversions. This particular table converts any sequence of |b…|r to use the Emphasized Text style.

```
c This converts text between |b and |r to emphasized text.

group(main)
'<Run ' > store(run) dup use(startRun)

group(startRun)
'>' > out(run) dup use(inRun)

group(inRun)
'</Run>' > dup use(main)
'|b' > '</Run>' nl out(run) ' namedStyle="Emphasized Text">'
'|r' > '</Run>' nl out(run) '>'
```

This table would convert

```
<Run ws="es">This is a |btest|r with |bemphasized text</Run>
```

to

```
<Run ws="es">This is a </Run>
<Run ws="es" namedStyle="Emphasized Text">test</Run>
<Run ws="es"> with </Run>
<Run ws="es" namedStyle="Emphasized Text">emphasized text</Run>
```

You could use similar changes to convert the bar code to some other writing system if that were desirable.

4.8 Splitting an interlinear text

Suppose you have an interlinear text and want to move some of the paragraphs to another text, or into a new text. There is no way to do this in the current program. You can cut some paragraphs from the baseline text and past them into a new text, but in the process all of the interlinearization, free translations, and notes get lost. However, this is fairly easy to fix in an XML dump file. The object is to move the desired StTxtPara elements to another text, making sure the id of the StTxtPara is not changed. The interlinearization, free translations, and notes will then move with the paragraph.

For example, consider the following interlinear text with two paragraphs. The second paragraph is highlighted. The goal is to move this paragraph to a new text and maintain the interlinearization, translations, and notes.

```
<Text id="ICDDDEA6F-82F6-40B9-B7EB-2C8700EEF348">
<DateCreated5><Time val="2006-08-15 09:19:26.003"/></DateCreated5>
<DateModified5><Time val="2006-09-28 14:15:34.507"/></DateModified5>
<Name5>
<AUni ws="en">Canoe trip</AUni>
</Name5>
<Description5>
<AStr ws="en">
<Run ws="en">From Anderson. 1897. Sena Grammar.</Run>
</AStr>
```

```
</Description5>
<Contents5054>
<StText id="I0FC1FC59-3751-4044-B062-CBD1A4B55D47">
<Paragraphs14>
<StTxtPara id="I7800AB04-D2FF-48B2-9D7D-3F8CBC6B1928">
<Contents16>
<Str>
<Run ws="seh">Wapakila m'mwadia, mbakwira pa mudi na maulo, dzuwa mbididoka;
mbatsama penepo.</Run>
</Str>
</Contents16>
</StTxtPara>
<StTxtPara id="I2AE1A969-EFDB-4D2E-919A-AB32C0FEE58D">
<Contents16>
<Str>
<Run ws="seh">Na masiku mbazidza nkhalamu ziwiri, mbazirira; mbagopa, mbapakila
pontho m'mwadia, mbawambuka ku nsua.</Run>
</Str>
</Contents16>
</StTxtPara>
</Paragraphs14>
</StText>
</Contents5054>
</Text>
```

Here is a template for a new interlinear text. The Name and Description elements are optional.

```
<Text>
<Name5>
<AUni ws="en">NewName</AUni>
</Name5>
<Description5>
<AStr ws="en">
<Run ws="en">NewDescription</Run>
</AStr>
</Description5>
<Contents5054>
<StText>
<Paragraphs14>
---- Insert StTxtParas here, making sure the id fields do not change ----
</Paragraphs14>
</StText>
</Contents5054>
</Text>
```

The solution is to add a new text from the template at the end of an existing Text, then move the highlighted paragraph to the new text. Interlinearization, translations, and notes all refer to the paragraph through the Id of the paragraph. So the essential thing is to make sure the Id is not changed in this process. Here’s how it looks when done.

```
<Text id="ICDDDEA6F-82F6-40B9-B7EB-2C8700EEF348">
<DateCreated5><Time val="2006-08-15 09:19:26.003"/></DateCreated5>
<DateModified5><Time val="2006-09-28 14:15:34.507"/></DateModified5>
<Name5>
<AUni ws="en">Canoe trip</AUni>
</Name5>
<Description5>
<AStr ws="en">
<Run ws="en">From Anderson. 1897. Sena Grammar.</Run>
</AStr>
</Description5>
<Contents5054>
<StText id="I0FC1FC59-3751-4044-B062-CBD1A4B55D47">
<Paragraphs14>
<StTxtPara id="I7800AB04-D2FF-48B2-9D7D-3F8CBC6B1928">
<Contents16>
<Str>
<Run ws="seh">Wapakila m'mwadia, mbakwira pa mudi na maulo, dzuwa mbidadoka;
mbatsama penepo.</Run>
</Str>
</Contents16>
</StTxtPara>
</Paragraphs14>
</StText>
</Contents5054>
</Text>
<Text>
<Name5>
<AUni ws="en">NewName</AUni>
</Name5>
<Description5>
<AStr ws="en">
<Run ws="en">NewDescription</Run>
</AStr>
</Description5>
<Contents5054>
<StText>
<Paragraphs14>
<StTxtPara id="I2AE1A969-EFDB-4D2E-919A-AB32C0FEE58D">
<Contents16>
<Str>
<Run ws="seh">Na masiku mbazidza nkhalamu ziwiri, mbazirira; mbagopa, mbapakila
pontho m'mwadia, mbawambuka ku nsua.</Run>
</Str>
```

```
</Contents16>
</StTxtPara>
</Paragraphs14>
</StText>
</Contents5054>
</Text>
```

Caution: If you want to move some segments but not the entire paragraph, the process is much harder since you would need to make sure all of the appropriate annotations, pointers, and offsets are set correctly. This is not recommended. See Conceptual model overview.doc Interlinear text section for more details.

4.9 Adding semantic domains to senses

Suppose you want to add semantic domains to senses based on the sense glosses. The following cc table will do this.

```
c Add semantic domains to senses
c This will add semantic domains based on the sense gloss, however, it will produce
c duplicate domains if the sense already contains that domain

c If any semanticDomains are stored, write them to the output and clear the storage
define(writeDomains) > ifneq(semanticDomains) ''
    begin
            '<SemanticDomains5016>' nl
            out(semanticDomains)
            '</SemanticDomains5016>' nl
            store(semanticDomains) endstore
    end

group(main)
'<LexSense ' > dup use(inSense)

c Active within a sense
group(inSense)
'</LexSense>' > do(writeDomains) dup use(main)
'<Gloss5016>' > dup use(inGloss)
'<SemanticDomains5016>' nl > append(semanticDomains) use(inDomains)

c Active within SemanticDomains within a sense
group(inDomains)
'</SemanticDomains5016>' nl > endstore use(inSense)

c Active in a gloss within a sense.
c Add one command for each gloss along with the domains to add for that gloss.
group(inGloss)
'</Gloss5016>' > endstore dup use(inSense)
'ws="en">to.understand<' > dup append(semanticDomains)
```

```
        '<Link ws="en" abbr="2"/>' nl
        '<Link ws="en" abbr="2.1.2"/>' nl
        endstore
    'ws="en">to.sleep<' > dup append(semanticDomains)
        '<Link ws="en" abbr="3.6"/>' nl
        endstore
    'ws="en">3PlObj<' > dup append(semanticDomains)
        '<Link ws="en" abbr="6.2"/>' nl
        '<Link ws="en" abbr="6.2.2"/>' nl
        endstore
```

CC groups provide an easy way to parse into an XML file to process given elements in specific ways. In this case group main copies everything unchanged until it gets to a sense. The inSense group is then active within a sense. XML end tags are a simple way to escape from an element and go back to the higher element. A gloss contains one or more AUni elements with each one specifying a different writing system. You could look for the entire element, but it can be abbreviated as in this example. The open angle bracket (<) is sufficient to identify the end tag since any literal < in the gloss would have to use the XML entity &lt;

Existing semantic domain links will have the full information from the dump file
<Link target="IBA06DE9E-63E1-43E6-AE94-77BEA498379A" ws="en" abbr="2" name="Person"/>
but the new items have the minimal elements that just specify the abbreviation.
<Link ws="en" abbr="2"/>
These shorter elements will work just as well since the import process will automatically find a semantic domain with this abbreviation and set the link to that domain. If an existing domain is not found, one will be created with this abbreviation.

When writing this kind of table, keep in mind that the sense may or may not already have some domains. To handle this, any existing domain links are appended to a semanticDomains storage area along with any new domain links coming from group(inGloss). At the end of the sense, if there are any stored domains, the domains are written out before the closing sense marker. This works because it doesn’t matter where the SemanticDomains5016 element is located within a <LexSense> element.

4.10 Adding pictures to senses

Suppose you want to add pictures to senses based on the sense glosses. The following cc table will do this.

```
    c Add pictures to senses
    c This will add pictures to senses based on the glosses.

    begin > store(fileId) '1' endstore

    c This writes all files at the end of the project.
    define(writeFiles) > ifneq(files) ''
       begin
          '<Pictures6001>' nl
```

```
        '<CmFolder>' nl
        '<Name2>' nl
        '<AUni ws="en">Local Pictures</AUni>' nl
        '</Name2>' nl
        '<Files2>' nl
        out(files)
        '</Files2>' nl
        '</CmFolder>' nl
        '</Pictures6001>' nl
        store(files) endstore
    end

c This writes all pictures at the end of a sense
define(writePictures) > ifneq(pictures) ''
    begin
        '<Pictures5016>' nl
        out(pictures)
        '</Pictures5016>' nl
        store(pictures) endstore
    end

c This appends a file name and caption in files and pictures
define(storePicture) > ifneq(fileName) '' c Assume we don't have captions without file names
    begin
        append(files)
        '<CmFile id="F' outs(fileId) '">' nl
        '<OriginalPath47><Uni>C:\Documents and Settings\Zook\My Documents\My Pictures\'
            outs(fileName) '</Uni></OriginalPath47>' nl
        '<InternalPath47><Uni>Pictures\' outs(fileName) '</Uni></InternalPath47>' nl
        '</CmFile>' nl
        append(pictures)
        '<CmPicture>' nl
        '<PictureFile48><Link target="F' outs(fileId) '"/></PictureFile48>' nl
        ifneq(caption) ''
         begin
            '<Caption48>' nl
            '<AStr ws="en">' nl
            '<Run ws="en">' outs(caption) '</Run>' nl
            '</AStr>' nl
            '</Caption48>' nl
        end
        '</CmPicture>' nl
        store(fileName,caption) endstore
        incr(fileId)
    end
```

```
group(main)
'<LexSense ' > dup use(inSense)
'<Pictures6001>' > use(inPictures) c Keep existing files
'</LangProject>' > do(writeFiles) dup

c This assumes there is only one CmFolder called Local Pictures.
group(inPictures)
'</Pictures6001>' nl > use(main)
'<CmFile ' > use(inFile) append(files) dup
'' > omit c Remove everything other than CmFiles

group(inFile)
'</CmFile>' nl > dup endstore use(inPictures)

group(inSense)
'</LexSense>' > do(writePictures) dup use(main)
'<Gloss5016>' > dup use(inGloss)
'<Pictures5016>' nl > append(pictures) use(inPictures)

group(inPictures)
'</Pictures5016>' nl > endstore use(inSense)

c Active in a gloss within a sense.
c Add one command for each gloss along with the picture(s) to add for that gloss.
group(inGloss)
'</Gloss5016>' > endstore dup use(inSense)
'ws="en">cow<' > dup append(pictures)
   store(fileName) 'Holstein cow.jpg' store(caption) 'Holstein cow' endstore do(storePicture)
   store(fileName) 'Longhorn steer.jpg' store(caption) 'Longhorn steer' endstore
do(storePicture)
'ws="en">duck<' > dup append(pictures)
   store(fileName) 'duck.jpg' store(caption) 'Duck' endstore do(storePicture)
```

This example is somewhat like the previous one, but involves creating new linked objects in the process. See section 2.2.7 in Conceptual model overview.doc for an overview of pictures in Fieldworks. In FieldWorks, CmFile objects are stored in a CmFolder named "Local Pictures". A separate CmFile object holds the original file name of the picture and the relative file name of the copied picture in %ALLUSERSPROFILE%\Application Data\SIL\FieldWorks. In a FieldWorks XML file this is represented as:

```
<Pictures6001>
<CmFolder id="IC2D69A90-2166-4A32-8EFC-080674F44A31">
<Name2>
<AUni ws="en">Local Pictures</AUni>
</Name2>
<Files2>
<CmFile id="IEB126F83-66F3-4599-B46A-E3AA59BACA5A">
<OriginalPath47><Uni>C:\Documents and Settings\Zook\My Documents\My
```

```
Pictures\Holstein cow.jpg</Uni></OriginalPath47>
<InternalPath47><Uni>Pictures\Holstein cow.jpg</Uni></InternalPath47>
</CmFile>
</Files2>
</CmFolder>
</Pictures6001>
```

A picture in a sense is represented by a CmPicture object which points to the CmFile object representing the picture and holds an optional caption. It is represented in a FieldWorks XML file as:

```
<Pictures5016>
<CmPicture id="IE333313E-4A4F-4D46-A768-6F20F9CD6494">
<PictureFile48><Link target="IEB126F83-66F3-4599-B46A-
E3AA59BACA5A"/></PictureFile48>
<Caption48>
<AStr ws="en">
<Run ws="xkal">Holstein cow</Run>
</AStr>
</Caption48>
</CmPicture>
</Pictures5016>
```

For the pictures to show up in Flex, they will need to be copied to your %ALLUSERSPROFILE%\Application Data\SIL\FieldWorks\Pictures directory. FieldWorks normally does this automatically when you insert a picture.

The cc table gathers any existing CmFile objects along with new ones generated by the storePicture define into a files storage area. Likewise, for each sense, it gathers any existing CmPicture objects along with new ones generated by the storePicture define into a pictures storage area. The storePicture define appends a new CmFile object to the files storage and CmPicture object to the pictures storage. An ID is not generated for the CmPicture since nothing links to it. The CmPicture is linked to the CmFile using a unique ID generated using the fileId number that is incremented after each use. At the end of each sense, any pictures are written out before the end sense marker. At the end of the language project, any files are written out before the end project marker. Several assumptions are made in this table. These assumptions are normally true, but if not, then changes would need to be made to the table.

- The table could accidentally add a new picture to a sense where that sense already has the same picture. The result would just be two copies of the same picture.
- The table assumes file names used in the inGloss group are unique and will not collide with existing files in the Pictures directory.
- The table will run into problems if the same picture file is used in multiple senses. What should happen in this case is the creation of one instance of CmFile for the picture, and multiple CmPictures pointing to that file. Less desirable, but legal, would be to create a new CmFile for each picture, but the InternalPath name and file will have to have a unique number appended to the end of the file name to avoid conflicts.

- The table assumes there is only one CmFolder in Pictures6001 and that it is named ‘Local Pictures’. This is all that the program will currently generate, but the conceptual model allows for more folders.
- The storePicture define assumes there will never be a caption without a picture file.