PharmaSUG 2014 - Paper BB11

XML in a SAS® and Pharma World

Mike Molter, d-Wise, Raleigh, NC

ABSTRACT

The introduction of standards to the clinical data life cycle has brought about significant changes to the job of a SAS programmer. Traditionally, SAS programmers who built data sets and statistical output for regulatory submission purposes had little need for technical knowledge beyond SAS programming and maybe some Microsoft or Adobe basics. In today's world, collected data can be made available in an XML format produced by EDC systems (i.e. ODM.xml). Submissions are expected to be accompanied by metadata expressed through an XML extension of ODM (i.e. Define.xml). There's even talk of replacing the traditional version 5 transport files with another XML extension of ODM (i.e. SDS-XML) for submission of domain data. The increased use of XML for transferring data and metadata has its advantages, but also imposes new requirements of a programmer's skill set. This paper serves as an introduction to some of these new skills. In it we'll discuss general XML basics and examine how they are applied in our industry today. We'll also look at tools that allow us to move data from a SAS data set to an XML format and vice versa. This paper is for industry SAS programmers at an intermediate level or above.

INTRODUCTION

The increasing prevalence of XML in today's clinical data life cycle brings with it a new set of skills required by SAS users that need to work with it. With operational CRF data being received and submission data and metadata being sent in this format, programmers now need to explore SAS functionality they may not have found necessary until now. Additionally, in order to use these tools effectively, these same programmers must become comfortable with XML basics. This paper will be split into two major sections. In the first, we'll take a brief tour through XML, at times comparing it to other languages such as SAS. Readers will become familiar with basic XML structure and the rules that guide it. In the second section we'll explore multiple SAS tools used to work with XML. Each tool will be covered at a high level, but in the end, the reader will have a basis from which to develop a deeper understanding with additional research and experimentation.

XML - THE WHATS AND WHYS

XML is a computer language that stands for Extensible Markup Language (we'll explain "Extensible" later). XML is used for storing and transporting data. Before we get too far into XML, let's first talk about markup language.

When we think of programming languages like SAS, we think of keywords, or special text contained in "programs" that act as "commands", and we think of processors that act on these commands when the text is "executed". In the case of SAS and other programming languages, these commands are executed to manipulate and/or analyze data that is stored in external files (e.g. data sets, tables). Markup languages are different. Markup languages are contained in Markup Files. Markup files contain both data (or content) along with keywords or "tags" that are recognized by processors as sets of instructions. One of the most popular markup languages is HTML. HTML files contain web page content that is displayed in a web browser according to HTML tags.

Aside from these fundamental differences, all computer languages, markup and programming, share a core set of characteristics. All are written in plain text, but the text and the way it is organized in each is subject to a set of rules that guide the structure and syntax of the text. Each language comes with its own vocabulary. In SAS we talk about PROCs, system options, and DATA steps. In XML, we talk about elements, attributes, tags, and schemas. Computer languages typically contain *keywords*, or special strings of text that have special meaning for the language. What *really* brings the language to life though is a processor. Without a processor, SAS programs and markup files are nothing more than text files with funny-looking words put together in a strange way. The processor reads the file, evaluates it for proper syntax, and executes its instructions. We can think of a SAS processor in action when we click the icon with the little running man (interactive mode) or when we submit a command from outside of the SAS environment (batch mode). An HTML file comes to life when we simply open it using a program called a web browser, which reads the HTML tags and displays the file's contents accordingly. With XML though, with regards to keywords and processors, the story changes.

XML STRUCTURE

Before we get too deep into the nuances of XML, we must first introduce some of the basics. First and foremost, XML files are made up of *elements*. An element is identified by its case-sensitive name, and an instance of an element can be detected by the presence of the element name surrounded by angle brackets.

```
<NHL>
```

This example illustrates the *start tag* of the *nhl* element. Elements are always defined by a pair of *tags*. The element begins with a start tag and ends with an *end tag*. The end tag looks like a start tag with a slash preceding the name of the element.

```
</NHL>
```

Elements may also contain any number of *attributes*. Attributes are properties of the element and are also identified by their case-sensitive name. They are provided as name-value pairs with quotation marks surrounding the value, and are specified alongside the element name in the start tag.

```
<team name="Red Wings">
```

What makes elements, or XML files as a whole, interesting is what comes between the start tag and end tag. Some elements contain data or *element content*.

```
<Conference>Eastern</Conference>
```

In the above example, the text "Eastern" makes up the content of an instance of the `Conference` element. Elements may also contain other elements.

```
<team name="Red Wings">
   <Conference>Eastern</Conference>
   <Division>Atlantic</Division>
   <Location>Detroit</Location>
</team>
```

In this example, the `team` element contains or *nests* three other elements - `Conference`, `Division`, and `Location`. Informally, we sometimes refer to the `team` element as a *parent* element, relative to the three elements it nests. We might also refer to the three nested elements as *child* elements or *sub-elements*. An XML element that is nested within another must be *properly nested,* meaning that its start and end tags must both be between the start and end tags of its parent element.

Occasionally an element will contain no content or nested elements. Such an element is said to be *empty*. Rather than including a start tag and an end tag, the following shorthand is allowed.

```
<element-name attributes />
```

Every XML file must contain a *root element*. A root element is an element whose start tag is at the beginning of the file and whose end tag is at the end. The following is an example of a complete XML file.

```
<nhl>
   <team name="Red Wings">
      <Conference>Eastern</Conference>
      <Division>Atlantic</Division>
      <Location>Detroit</Location>
   </team>
   <team name="Flames">
      <Conference>Western</Conference>
      <Division>Pacific</Division>
      <Location>Calgary</Location>
   </team>
   <team name="Devils">
      <Conference>Eastern</Conference>
      <Division>Metropolitan</Division>
      <Location>New Jersey</Location>
   </team>
</nhl>
```

Although XML has a few more rules around syntax and structure, those mentioned above cover most of what we need to know for this paper. The example above is said to contain *well-formed* XML. Well-formed XML simply refers to XML that follows these basic syntax and structure rules.

We mentioned above that data in an XML file makes its appearance between the start and end tags of XML elements. For that reason the reader should not be alarmed to find text related to NHL hockey teams between the start and end tags in the example above. But upon closer inspection we find hockey terms in more than just the data. We find it not only in attribute values but also in attribute names. Maybe what's most surprising is that we find hockey

terms in element names. Earlier we noted that keywords are a fundamental part of computer languages. Those that have even an elementary understanding of HTML know that in HTML, element names serve as keywords and are processed by web browsers as instructions for displaying content. Are we suggesting that XML is a computer language whose sole purpose is to process hockey data? Not exactly.

GENERAL XML VS SCHEMA XML

Where XML starts to distinguish itself from other computer languages is in the fact that it has no pre-defined keywords. Of course keywords are keywords, meaning they have special meaning in a language, because of how they're interpreted by a processor. XML's lack of keywords stems from the fact that XML has no general processor! Imagine SAS without a processor - lots of text, including "words" you won't find in a dictionary, with lots of semicolons distributed throughout, along with occasional other characters like commas and equals signs. Without a processor, XML developers are free to make up whatever element names and attribute names they wish. This fact means that the contrived example above, from a syntactical, a structural, and overall "legal" perspective, is just as much of an XML document as any other document used with any software in the world today. Of course the natural question is, without a processor, what good is it?

We can illustrate the usefulness of XML by introducing the evolution of an XML *schema* from general XML or XML not associated with a schema using the hockey example. Let's suppose that the NHL (National Hockey League, the highest level of professional hockey in North America) keeps all of its data, including that which is contained in the example above, in a proprietary database (e.g. SAS, Oracle, Microsoft Access, etc.). The NHL also has tools that allow its analysts to query, manipulate, and analyze the data in whatever way is needed. By itself, this doesn't seem to be a problem, but now let's suppose that the NHL realizes a need to exchange data with other hockey leagues, such as the American Hockey League (AHL - professional league just below the NHL), the National Collegiate Athletic Association (NCAA - college hockey), and the Olympics. The challenge in sharing data between leagues comes in the fact that they all store their data in different ways in different proprietary database systems. This means that queries and analyses to obtain information in one system may have to be executed differently to get the same information about another league in another system. It also means that analyses that include comparisons of players across leagues become almost impossible. What is needed is one non-proprietary format that all leagues can use. Each might develop its own software and tools for processing such a format, but this can only be done when technical personnel in each league get together, agree on all details of the syntax and structure of such a format, and devise a formal mechanism for enforcing such rules. This is where XML starts to become useful. By deciding to use XML, the technical developers now take on the task of determining not only the names of elements and attributes, but also rules around how they're used, how many instances of each is allowed, which elements are nested within which other elements, and many more. What the developers are developing is what is known as an XML *schema*.

Informally, we use the term *schema* to refer to a specific set of elements, attributes, and the rules that govern them that are intended for a specific purpose or use of XML. To a degree, we can start to think of XML schemas as containing keywords, since we have now defined specific elements and attributes that the XML documents must have - something that was missing from general XML. But the question still remains out there - just because some developers have gotten together, agreed on some keywords and shook hands, what good are they without a processor? Can we still call them keywords? Is there anything that can prevent us from breaking these rules?

The story doesn't end with the handshakes. Keep in mind that the purpose was to create a format that could then be consumed by each of the leagues. Once the meeting is over, developers from each of the leagues go back to their offices and begin to design tools that can process XML files that abide by the agreed upon format. In order for a piece of software to take advantage of the keywords and the rules that were agreed upon, it first has to know that the XML file is playing by the rules. Put another way, it has to know that the XML file is *valid* with respect to the rules. For this reason, the rules must be documented in a file that an *XML validator* can use.

As mentioned above, we sometimes use the term "schema" to informally refer to a set of rules, which includes necessary elements and attributes, allowed frequency of elements and attributes, structure, and others, which governs a specific XML use. The term XML *vocabulary* is used in the same way. More formally though, an *XML schema file* is a file that an XML validator can use that explicitly documents these rules. Several types of schema files exist. One is the Document Type Definition or DTD, which defines the legal set of elements and attributes. An alternative to the DTD is the XML Schema Language or XML Schema Definition (XSD). XSD is written in XML and so obeys the structural rules described above. XSD is also more powerful than the DTD, with the ability to be more specific in its description of an XML structure. We often say that an XML file that obeys the rules of a schema is *schema-valid*.

If it helps, think of how the SAS processor works. When you execute a DATA step or a PROC, the processor first browses the code you're executing to make sure it understands what you're asking for. We think of this as the *compiling* stage. An XML processor may or may not be built with schema-validation, but plenty of software is built for the explicit purpose of schema validation. Just as a SAS processor will report on missing semicolons, un-closed DO

blocks, or mismatching parentheses before it even tries to carry out instructions, the XML schema validators will make sure the XML file is playing by the rules of the schema file you provide.

Schema files are part of what make XML extensible. While an XML schema may simply define explicitly its own set of elements and attributes, it may also on top of that import the elements and attributes of another schema. In other words, one schema may *extend* another schema.

What really puts the X in XML is the fact that an XML file might just use the elements defined in one schema, but it doesn't have to. The schema file makes sure that the document has certain elements and attributes and that obey certain rules, but it doesn't say anything about elements and attributes it doesn't know. This means that an XML file might contain elements and attributes from one vocabulary that was developed for a particular purpose, plus elements and attributes from another vocabulary, and so on. Additionally, an XML document might contain elements and attributes that nobody's ever heard of before.

The set of elements and attributes associated with a particular XML vocabulary is referred to as a *namespace*. A namespace is uniquely identified with a *uniform resource indicator* (URI) - a string of characters used for identifying an internet resource. Because in theory two namespaces may coincidentally use the same element or attribute name for different purposes, the use of an element from a namespace must be qualified with a prefix, separated from the element name with a colon, associated with that namespace. The prefix is established in the `xmlns` attribute that declares a namespace.

XML extensibility often becomes convenient when the nature of data is similar to that for which an XML schema already exists. Returning now to the meeting involving technical personnel from each of the hockey leagues, imagine that in developing an XML schema, the NCAA shares with everyone the XML schema that they use to store and exchange data across all collegiate sports. The suggestion is to use this schema as a starting point rather than developing a new one from scratch. Upon reviewing this schema, developers realize both that not all elements and attributes from the NCAA schema will be needed, and that additional elements and attributes not in the NCAA schema will need to be defined. The latter set will make up a new namespace.

In a small way, we can see similarities between the use of namespaces in an XML document and library references (i.e. librefs) in a SAS program. In a SAS program, we declare a pointer to a SAS library with a LIBNAME statement. This pointer is then used as a prefix to data set names in that library. Two data sets with the same name may exist in two different libraries, but this prefix distinguishes between the two. In an XML document, we declare namespaces using the `xmlns` attribute on any element whose child elements we may want to use the prefix. Oftentimes namespaces are declared in the root element.

```
<root-element
   xmlns:ncaa="ncaa-URI"
   xmlns:hky="hockey-URI"
```

After this point, elements from the NCAA namespace are prefixed with "ncaa:", and elements from the new namespace developed by personnel from the various hockey leagues are prefixed with "hky:". This is especially convenient if the two namespaces coincidentally have elements of the same name but otherwise have nothing to do with each other. For example, the NCAA schema might use the element `Penalty` to store information about NCAA violations (in any sport). The Hockey schema may have no use for this data, but it may have use for storing data about penalties that occur during hockey games. Declaring a namespace with an associated prefix and using this prefix with each instance of an element in this namespace allows processors to understand how elements are to be processed.

```
<hky:penalty name="hooking">
   <hky:minutes>2</hky:minutes>
</hky:penalty>
```

With the concepts of XML schemas, namespaces, and general extensibility, the hockey leagues now have everything we think of when we think of computer languages. Along with structural and syntax rules, the existence of a schema file now allows IT personnel to build processors that can count on the existence of keywords in the form of elements and attributes defined in the schema, and act accordingly. Just as SAS reports configuration errors in its log when keywords are used improperly, so too can an XML validator when the keywords are not used according to the schema.

On the one hand, the hockey example was contrived for the purpose of illustration within this paper. However its contrived nature serves to illustrate an important feature of XML. There may not be an XML schema used by the NHL and other hockey leagues to exchange data, but there could be. XML schemas can be thought of as computer languages that anyone can create for the purpose of data exchange using simple text documents. Of course access to resources with the knowledge of building processors that can give more meaning to a schema helps. The following is a list of common XML schemas in widespread use across industries today.

- XML Schema (XSD) - as noted above, this is one type of XML file validator. XML Schema itself is a language that obeys all of the syntax and structure rules of XML. Additionally, it has its own set of required elements and attributes, and XML validators process such files, giving meaning to its keywords.
- Extensible Stylesheet Language (XSL) - XSL is an XML schema used by processors to transform XML files into other text-based documents such as HTML or XML files. We'll see examples of this language later in the paper.
- XML Spreadsheet 2003 - This is an XML schema used by Microsoft Excel to read data and display as Excel spreadsheets. To see an example, open an Excel file, choose Save As, click on the dropdown arrow next to Save As Type and choose XML Spreadsheet 2003. Then use a text editor to open the same file.
- RSS - Allows web developers to offer personalized views of their web page using RSS viewers

XML SCHEMAS IN THE PHARMACEUTICAL INDUSTRY

The XML schema off of which all schemas related to clinical and non-clinical trial data in the pharmaceutical and biotech industries are based is referred to as the Operational Data Model, or ODM. Developed by the XML Technologies team of the Clinical Data Interchange Standards Consortium (CDISC), ODM is meant to store and exchange collected clinical trial data, along with metadata that describes the collected data, administrative data, reference data, and audit information. Applying the principals of extensibility discussed above, the team has also used this model as a basis for defining a schema for the exchange of submission metadata (Define-XML). ODM has also been extended to define a schema used for submission data (Dataset-XML). For the rest of this section we'll discuss a few of the high points of ODM, both the elements and attributes as well as structure. We'll also see how Define-XML extends the ODM schema. At the time of this writing, Dataset-XML is still in review and being piloted by FDA. For that reason we'll keep discussion of this schema at a minimum.

In ODM we see the word "item" used as part of the name of many elements. We can think of ODM items as variables in collected data. We'll also see "ItemGroup" used frequently. We can think of an item group as a group of variables, or a data set. Element name suffixes are also important, and one of the most common suffixes is "Data". At a high level, the `ReferenceData` element is used to contain non-subject reference data such as information about lab ranges. At the same level, the `ClinicalData` element is used to contain subject data. Both of these elements contain within them one `ItemGroupData` element for each observation (think of row of a data set), which contains an `ItemData` element for each variable. It is these `ItemData` elements that contain data values.

Commonly found in the metadata portion of an ODM file are elements whose name end in "Def" or "Ref". "Def" is short for "definition". An `ItemGroupDef` element defines metadata for a data set, while `ItemDef` elements define metadata for variables. "Ref" is short for "reference". These elements declare an instance of something without actually defining it. For example, contained within the `ItemGroupDef` is one `ItemRef` for each variable in the data set. These references declare instances of variables in the data set.

In addition to defining data sets and variables, ODM metadata also defines *codelists*, or lists of terms that define the allowable values of one or more variables. Each codelist is defined in a `CodeList` element (not sure why it's not CodeListDef). When a variable has a set of allowable values, the `ItemDef` that defines the variable contains a reference to that codelist in the form of a `CodeListRef` element.

ODM is also filled with attributes informally referred to as OIDs. OID stands for Object Identifier. While the value of these attributes is often subject to a sponsor's convention, they generally have no meaning to the data. The purpose of OIDs is to link different pieces of the ODM together using matching values. OID attributes are either named "OID" or are suffixed with "OID". Attributes named OID are generally found in metadata definition elements (e.g. `ItemGroupDef`, `ItemDef`, `CodeList`). The other OID attributes are usually found in elements that refer to these definitions and are prefixed by what is being referenced and defined. The definition of an ODM entity such as an item, whose OID value matches that of an instance of that item, establishes a unique link between an instance and a definition in much the same way that a section of text in a book is linked to an explanation in a footnote or an end note by way of a superscripted asterisk or letter.

Figure 1 below is a small excerpt of an ODM file and illustrates many of the features discussed above.

```
<ClinicalData StudyOID="STUDY.StudyOID" MetaDataVersion="v1.1.0">
 <SubjectData SubjectKey="001">
  <StudyEventData StudyEventOID="SE.VISIT1" StudyEventRepeatKey="1">
   <FormData FormOID="FORM.AE" FormRepeatKey="1">
    <ItemGroupData ItemGroupOID="IG.AE" ItemGroupRepeatKey="1">
     <ItemData ItemOID="ID.TAREA" Value="ONC"/>
     <ItemData ItemOID="ID.PNO" Value="143-02"/>
```

```
      <ItemData ItemOID="ID.SCTRY" Value="USA"/>
      <ItemData ItemOID="ID.F_STATUS" Value="V"/>
      etc.
</ClinicalData>

<ItemGroupDef OID="IG.AE" Repeating="Yes"
 SASDatasetName="AE"
 Name="Adverse Events"
 Domain="AE"
 Comment="Some adverse events from this trial" >
  <ItemRef ItemOID="ID.TAREA"    OrderNumber="1" Mandatory="No"/>
  <ItemRef ItemOID="ID.PNO"      OrderNumber="2" Mandatory="No"/>
  <ItemRef ItemOID="ID.SCTRY"    OrderNumber="3" Mandatory="No"/>
  <ItemRef ItemOID="ID.F_STATUS" OrderNumber="4" Mandatory="No"/>
  etc.
</ItemGroupData>

<ItemDef OID="ID.TAREA" Name="TAREA" DataType="text" Length="20">
 <Description>
  <TranslatedText xml:lang="en">TAREA Label</TranslatedText>
 </Description>
</ItemDef>

<ItemDef OID="ID.PNO" Name="PNO" DataType="text" Length="10">
 <Description>
  <TranslatedText xml:lang="en">PNO Label</TranslatedText>
 </Description>
</ItemDef>

<ItemDef OID="ID.SCTRY" Name="SCTRY" DataType="text" Length="15">
 <Description>
  <TranslatedText xml:lang="en">SCTRY Label</TranslatedText>
 </Description>
</ItemDef>
```

Figure 1

We first note from Figure 1 that we are defining data for an individual subject. The `ItemGroupData` element represents data for a group of items, or variables. Its `ItemGroupOID` attribute serves as a reference or a pointer to the definition of this item group or its metadata. Note that its value, IG.AE, matches that of the value of the OID attribute in the definition of the item group, the `ItemGroupDef`. In this definition we learn about the name, label, and other attributes of the data set. Within `ItemGroupData` we see `ItemData` elements where we find variable values in the `Value` attribute. The other attribute of `ItemData` is `ItemOID`, a pointer to the definition of the item, or `ItemDef`. We also note that the `ItemRef` elements within `ItemGroupDef` also contain the same `ItemOID` attributes with the same values. That is because within the context of a data set, references to items that are contained in that data set are appropriate, but because a variable can appear in multiple data sets, defining a variable must take place outside the context of a data set. For that reason `ItemDef` elements appear outside of `ItemGroupDef` elements. They contain OID attributes whose values match the `ItemOID` values found in item group definitions as well as item group data.

Define-XML is an XML schema used for the storage and exchange of the metadata that describes the trial data submitted to the regulatory agency. We know that ODM contains elements used for the metadata that describes the collected data. Although the data being described by ODM's metadata is different from the data being described by Define-XML, it was decided that the ODM metadata elements could be also used in Define-XML. For that reason, Define-XML uses ODM metadata elements such as `ItemGroupDef`, `ItemDef`, and `CodeList`. The following is a declaration of the ODM namespace often found in an attribute of the ODM element (the root element of ODM and Define-XML).

```
xmlns="http://www.cdisc.org/ns/odm/v1.3"
```

It's important to note in the declaration above that no prefix is defined for this namespace (recall that the prefix follows xmlns: in the attribute name). This fact makes this the default namespace. When a default namespace is defined, elements without a prefix come from this namespace.

Define-XML does have its own namespace for metadata elements not relevant in ODM. Among these are elements for *value-level metadata*. Value-level metadata is used to describe certain variables in submission data whose metadata depends on the values of another variable. The need for value-level metadata is a result of the vertical nature of certain data structures such as the Basic Data Structure (BDS) in ADaM, as well as Findings and Supplemental Qualifier (SUPPQUAL) structures in SDTM. Metadata for AVAL in the BDS structure depends on PARAMCD; --ORRES in the Findings structure depends on --TESTCD; and QVAL in the SUPPQUAL structure depends on QNAM.

The declaration of the Define-XML namespace often looks like this.

```
xmlns:def="http://www.cdisc.org/ns/def/v2.0"
```

We note here that "def" is the prefix associated with elements and attributes in the Define-XML namespace. Value-level metadata starts with a reference to it from an ItemDef element that defines --ORRES, AVAL, or QVAL.

```
<ItemDef attributes >
  <def:ValueListRef ValueListOID="VL.VSORRES"/>
  other child elements of ItemDef
</ItemDef>
```

Of course we know that wherever we see a reference element like `ValueListRef`, somewhere there must be an associated definition element, linked together by OID attributes.

```
<def:ValueListDef OID="VL.VSORRES">
 <ItemRef ItemOID="IT.VS.VSORRES.DIABP" OrderNumber="1" Mandatory="Yes">
   <def:WhereClauseRef WhereClauseOID="WC.VS.VSTESTCD.DIABP"/>
 </ItemRef>
 <ItemRef ItemOID="IT.VS.VSORRES.HEIGHT" OrderNumber="3" Mandatory="Yes">
   <def:WhereClauseRef WhereClauseOID="WC.VS.VSTESTCD.HEIGHT"/>
 </ItemRef>
etc
```

The value list definition is a series of item references (`ItemRef`) like the ones we saw above within the item group definitions (`ItemGroupDef`). We know that item references are references to variable definitions, but the item references in value list definitions refer to what we might think of as *virtual variables.* The first one in the example above references metadata that describes collected results for diastolic blood pressure and the second references

metadata that describes collected results for height. We refer to these as virtual variables because although they aren't real variables in the SDTM submission data sets, conceptually they are variables in the sense that they all have their own metadata. The collected results for diastolic blood pressure can be described with metadata that differs from that that describes collected results for height. The fact that they are placed in a vertical structure doesn't change this fact. For that reason, each of these virtual variables has its own item definitions.

Define-XML has a few other elements that aren't in the ODM namespace. One of them is seen in the previous example. The Where clause is new to version 2.0 of Define-XML. As with other elements, we have a Where Clause reference (`def:WhereClauseRef`) and a Where Clause Definition (`def:WhereClauseDef`) which explicitly defines the condition under which the item definition applies (e.g. VSTESTCD="DIABP"). Other examples include `def:Origin` which describes the origin of a submitted variable (e.g. the page number of an annotated CRF), and `def:CommentDef` which contains comments at a data set, variable, or value level.

Since Dataset-XML was not yet in production at the time this was written, we won't get into too many details about it. In short, just as Define-XML used the metadata elements of ODM, Dataset-XML uses a subset of the data elements of ODM. While the data in ODM represents collected data, Dataset-XML represents submission data, either SDTM or ADaM.

At this point we've now seen the basics of XML. We've seen that general XML differs from other computer languages in that it has no keywords because there's no one XML processor to define such keywords. Put another way, we've seen that meaningful XML vocabularies are custom built for particular purposes, along with processors that act on them and schemas to validate them. Finally, we've seen the way that CDISC has built a master XML structure called ODM as well as others that are built on ODM for the storage and exchange of clinical data and metadata. Of course SAS programmers are accustomed to working with this data when it's stored in SAS data sets. If the data arrives to us in XML or has to be submitted for review in XML, then the next step is to understand how we can easily move our data between the two formats. Several options exist to accomplish this.

SAS TOOLS FOR READING AND WRITING XML

In this section we'll look at several tools for reading and writing XML. We'll start with tools that are easy to use but only read and write specific structures, sometimes with specific tags. We'll then move to more customizable solutions which sometimes require expanding our skill sets. In those instances we'll take a quick peek at some new technologies but will leave it to the reader to explore them further.

SAS SOLUTIONS OUT OF THE BOX

THE XML ENGINE

We'll start by looking at the XML engine of the LIBNAME statement. Typically when we think of the LIBNAME statement, we think of access gained to a file directory by way of an associated pointer named with eight characters or less. With this access we can manipulate and analyze data sets inside these directories as well as move data sets into and out of them. With this in mind, and knowing that an XML file is just that - a file - it might be hard to imagine how this statement is used. But it might be a little easier if we remember what XML is for - storing data. With the XML engine, we establish a pointer to an XML *file*. If we're writing XML, we might point to an XML file that doesn't yet exist until we write into it with typical SAS functionality. If we're reading XML then we point at the file we're reading.

The following LIBNAME statement establishes a pointer to a non-existent XML file.

```
libname out1 xml file='C:\simple_file.xml' ;
```

We first note the use of the XML engine between the libref (OUT1) and the file specification. Executing this statement does not create the file, but it does allow us to create it with traditional data set creation tools. The following example creates the file with data from SASHELP.CLASS using PROC COPY.

```
proc copy in=sashelp out=out1 ;
select class ;
run;
```

The DATA step is another option.

```
data out1.class ;
set sashelp.class ;
run;
```

Below is an excerpt from simple_file.xml, using either PROC COPY or the DATA step.

```
<TABLE>
   <CLASS>
      <Name> Alfred </Name>
      <Sex> M </Sex>
      <Age> 14 </Age>
      <Height> 69 </Height>
      <Weight> 112.5 </Weight>
   </CLASS>
   <CLASS>
      <Name> Alice </Name>
      <Sex> F </Sex>
      <Age> 13 </Age>
      <Height> 56.5 </Height>
      <Weight> 84 </Weight>
   </CLASS>
```

Compare the output above to the first two observations of SASHELP.CLASS.

	Name	Sex	Age	Height	Weight
1	Alfred	M	14	69	112.5
2	Alice	F	13	56.5	84

We note that the LIBNAME engine produced a file in which TABLE is the root element. Within TABLE, we have repeating instances of an element called CLASS. Each instance represents an observation of the data set. Within each instance are single instances of elements that each represent a variable of the data set. The value of these elements represents the value of the data set variable for that observation.

By default this is the structure that the LIBNAME engine produces - a root element called TABLE, repeating instances of an element named for the data set being placed into the document, and within each instance, elements named for the variables whose values are the values of the variables. The LIBNAME statement does offer the XMLTYPE= option that creates slightly different structures depending on its value (e.g. XMLTYPE=ORACLE or XMLTYPE=MSACCESS).

An XML file that is in the structure seen above can also be imported into a SAS data set. Likewise, files that follow the structure produced with the XMLTYPE option can be imported with the same option. In the following example, we copy data from the XML document to the WORK library. CLASS appears in the SELECT statement because with CLASS appearing as the repeating element underneath the root element, the XML engine recognizes this as the name of the data set.

```
libname out1 xml file='C:\simple_file.xml' ;
proc copy in=out1 out=work ;
select class ;
run;
```

The XML engine does also support the import and export of multiple data sets in the basic element structure above. Each unique child element of the root element represents the name of a corresponding unique data set. The following excerpt is read by the XML LIBNAME engine as two separate one-observation data sets - MALE and FEMALE.

```
<TABLE>
   <MALE>
      <Name> Alfred </Name>
      <Sex> M </Sex>
      <Age> 14 </Age>
      <Height> 69 </Height>
      <Weight> 112.5 </Weight>
   </MALE>
   <FEMALE>
      <Name> Alice </Name>
      <Sex> F </Sex>
      <Age> 13 </Age>
      <Height> 56.5 </Height>
      <Weight> 84 </Weight>
```

```
</FEMALE>
```

We can then copy the data into two WORK data sets.

```
proc copy in=out1 out=work ;
select male female ;
run;
```

We can also export two data sets to an XML file, but only using PROC COPY (the DATA step doesn't work).

TAGSETS

The mechanism that sits behind the scenes that is responsible for the XML markup produced by the LIBNAME engine is the SAS *tagset*. A tagset is a template that guides the way that markup output is produced. Tagsets are specifically designed to facilitate the building of a structure that involves nested elements such as XML. SAS comes with several pre-defined tagsets that are stored within the Tagsets directory of the TMPLMST item store in the SASHELP library.

The choice of a value for the XMLTYPE option on the LIBNAME statement is a choice of a tagset. When we use the XML engine without specifying a value for XMLTYPE, then by default, the value of XMLTYPE is GENERIC. This and other values translate into the use of pre-defined SAS tagset that governs the structure of the output file. In addition to the XMLTYPE option, users also have the TAGSET= option.

```
libname out2 xml 'C:\access_xml.xml' tagset=sasxmacc2003 ;
```

ODS

We know that the LIBNAME statement allows us access into SAS directories, which then allows the transfer of SAS data between directories. Because the exchange of data is a primary purpose of XML, it makes sense that an XML engine is available to the LIBNAME statement to facilitate this data movement. However it's also conceivable that PROC output that is often sent to formats such as HTML, PDF, RTF, and even plain text, may also need to be sent to an XML file. For this reason, an alternative to the LIBNAME XML engine is the Output Delivery System (ODS).

We send PROC output to an ODS destination by wrapping the PROC code in an "ODS sandwich" that defines the output format and specifies the destination. The following illustrates the use of the XML destination.

```
ods xml file='C:\xmldest.xml' ;
proc print noobs data=sashelp.class;
run;
ods xml close ;
```

Figure 2 illustrates an excerpt of output.

```
- <header name="Height" label="Height" type="string"
    class="header" unformatted_type="string"
    unformatted_width="6" row="1" column="4">
    <label>Height</label>
    <value>Height</value>
  </header>
- <header name="Weight" label="Weight" type="string"
    class="header" unformatted_type="string"
    unformatted_width="6" row="1" column="5">
    <label>Weight</label>
    <value>Weight</value>
  </header>
 </row>
</output-head>
- <output-body>
 - <row>
  - <data name="Name" label="Name" type="string"
    class="data" precision="0" scale="0"
    unformatted_type="string" unformatted_width="8"
    row="2" column="1">
    <label>Name</label>
    <value>Alfred</value>
  </data>
  - <data name="Sex" label="Sex" type="string"
    class="data" precision="0" scale="0"
    unformatted_type="string" unformatted_width="1"
    row="2" column="2">
    <label>Sex</label>
    <value>M</value>
```

Figure 2

The meat of this output contains two main sections - an `output-head` element which contains header information including column names and labels, and an `output-body` element which contains data in the `value` element. Both sections contain a `row` element to define observations.

Like the XML engine of the LIBNAME statement, we can also specify the name of a tagset for ODS to use to create markup. This is accomplished by using the TAGSET= option on an ODS statement that opens the MARKUP destination. The following statement generates XML that MS Excel can read and display as the spreadsheets we're accustomed to seeing in Excel.

```
ods markup tagset=excelxp file='C:\toexcel.xml' ;
proc print noobs data=sashelp.class;
run;
ods markup close ;
```

CUSTOMIZED SOLUTIONS

Up to this point, all of the methods for moving data between SAS and XML have focused on out-of-the-box options such as options on the LIBNAME statement and tagsets that SAS has defined. The advantage to such methods is the ease of use. The disadvantage is a big one - being confined to the specific tags and XML structures that those options define. In this section we examine solutions that allow us to have more control over how SAS both produces XML as well as how it reads it. The tradeoff is that in some cases, we have to dip our toes into some new technologies. In those cases, we'll address high-level concepts but will refer the reader to other resources for more information.

THE DATA STEP

We'll start with a simple solution for exporting SAS data that most SAS programmers should be familiar with. Because XML is nothing more than plain text, one way to produce XML is with the DATA step and simple PUT statements. Suppose the data set HOCKEYTEAMS is structured as below in Figure 3.

	name	conference	division	location
1	Red Wings	Eastern	Atlantic	Detroit
2	Flames	Western	Pacific	Calgary
3	Devils	Eastern	Metropolitan	New Jersey

Figure 3

The following DATA step reads the data set above and produces the NHL XML illustrated in the beginning of this paper.

```
filename xmlout4 'C:\teams_datastep.xml' ;
data _null_ ;
file xmlout4 ;
set hockeyteams end=thatsit ;

if _n_ eq 1 then put '<nhl>' ;
put '<team name="' name '">' ;
put '<conference>' conference '</conference>' ;
put '<division>' division '</division>' ;
put '<location>' location '</location>' ;
put '</team>' ;
if thatsit then put '</nhl>' ;
run;
```

In this example we use the familiar FILENAME statement to establish a pointer to a file, and then reference that pointer in the FILE statement within the DATA step. We use the END= option on the SET statement which creates a temporary variable (in this case, THATSIT), that allows us to identify the last record in the data set being read. Combining this with the identification of the first record read (if _n_ eq 1) allows us to easily generate the start tag and end tag of the root element. With each observation that is read, element and attribute names as well as characters such as equal signs, angle brackets, and quotation marks are hard-coded, while data is captured from the data set. Using DATA step tricks such as array processing, one could quickly modify the above code to make it more robust, allowing for any number of variables and any variable names.

The structure of the NHL XML file is close to the structure that is produced by the default LIBNAME XML engine. Two exceptions are that with the DATA step, we were able to add an attribute to the `team` element, and we were able to name the repeating `team` element the way we wanted to. The default behavior would have named it by the data set name, and would not have added an attribute. It should be clear that with the DATA step, we have control over all of the text written.

With the flexibility of the DATA step, why do we need any other tools for exporting XML? Technically, we can get away with a DATA step solution for any XML we need to produce, but as the XML gets more complicated, so too does not only the DATA step code, but also the structure of the data set being read. As an example, we need look no further than ODM and DEFINE-XML. We know that one such file contains several sections with a variety of information. One section contains administrative data, another contains clinical data, another contains metadata and another contains controlled terminology. While the data in these sections is not un-related to each other, it might not make sense to try and fit it all together in one data set to read. Even if one did choose this method, the other challenge would be proper management of start and end tags. The simple structure of the NHL structure above included only three simple levels of elements - the root element, a single repeating-instance element in which each instance represented an observation, and below that, elements that each represented a data set variable. The root

element was easily managed with straightforward identification of the first and last records being read. The rest was written with each record read. ODM has several levels of elements. Some of these elements have attributes while others don't. Some have element values. Some nest other elements beneath them. It isn't impossible, but with different elements and sections of elements containing their own unique structure, the DATA step approach gets unmanageable in a hurry.

CUSTOM TAGSETS AND THE EVENT MODEL

In addition to the ability to use tagsets built by SAS, users can also write their own tagsets. This means that users can customize the way that text is written to an output file. Most notably, tagsets provide a convenient model that facilitates the management of nested markup output. The challenge is in understanding how this model works.

Because tagsets are templates, they are built with PROC TEMPLATE. Other than a few statements that set attribute values for the tagset, most of what is found in a tagset template is *event* definitions. An event definition contains code to be executed when the event is *called*. Although the code has familiar functionality, some of the statements have a slightly different look than the SAS code to which we're accustomed. One familiar statement found in the *event model* that is responsible for the writing text to an output file is the PUT statement.

To make a long story short, when a tagset is being used to create output, SAS calls a series of events. For example, the opening of an ODS markup destination triggers SAS's calling of events that only ends when the destination is closed. When an event is called, data is passed through ODS by way of *event variables*. In the first several events, this data is mostly metadata. After that, data is passed through in the order in which it needs to be written to the output file. After some initial metadata, stylesheet data that governs display aspects of the file such as fonts and colors is passed. After that, the data that comes from the PROC executed in the ODS sandwich is passed. At this point it is passed in an order dictated by the way the PROC was written. For example, if the ODS sandwich contains a PROC PRINT with a VAR statement that lists the variables AGE, SEX, and HEIGHT in that order, then an AGE value will be passed, followed by a SEX value and then a HEIGHT value.

There are at least three significant challenges that one must overcome in order to feel comfortable with writing custom tagsets. The first is the use of the tagset language. Although PUT works exactly like it does in the DATA step, other functionality is accomplished in different ways. For example, variable assignment is accomplished through either a SET statement or an EVAL statement. Conditional statements are constructed with the condition *following* the action rather than preceding it. Other statements such as TRIGGER and BREAK are unique to the event model. These are documented in the SAS Online documentation. The second challenge is getting to know the names of the events and when they are called. The third is knowing the names of the event variables and what kinds of values they hold.

While documentation does exist on the tagset language, understanding the events and their variables isn't always as straightforward. Luckily we do have some tricks. When we write a DATA step that reads data, we have ways of knowing what the variables are in the data set we're reading. With the event model, we know that data is somehow passed through to ODS through variables, but writing code that processes it is like writing a DATA step that reads a data set that we have no way of knowing anything about. Oddly enough, we can find out more about the event model through certain SAS-supplied tagsets.

The tagsets EVENT_MAP and TEXT_MAP are examples of what are known as *mapping tagsets*. A mapping tagset clues us in on the events that are being called as well as their variables and the values they hold. Both of these mapping tagsets write the same information in two different ways. EVENT_MAP writes this information in an XML format where the tag names are the names of the attributes, and the attributes of these tags are the name-value pairs of many of the event variables. TEXT_MAP writes simple text and so can be directed to a simple .txt file and makes careful use of line spacing, indenting, and comments to explain the kind of data that is being passed through events. Figure 4 below is an excerpt from the beginning of an output file from a PROC PRINT of the hockey data above produced with the TEXT_MAP tagset.

```
/*-------------------------------------------------------
/*-- before we do anything we've got to write out the
/*-- beginning of the doc.
/*-------------------------------------------------------

Event Name is: doc
    Arguments:
       operator       "mmolter"
       sasversion     "9.3"
       saslongversion "9.03.01M1P11022011"
       date           "2014-02-03"
       time           "21:05:23"
       encoding       "windows-1252"
       state:          "start"
       trigger_name     "attr_out"
       total_page_count "0"
       page_count       "0"
       proc_count       "0"
       total_proc_count "0"
       class            "body"
       just             "1"

  /*-------------------------------------------------------
  /*-- The document head section.  A place for general
  /*-- definitions.
  /*-------------------------------------------------------

  Event Name is: doc_head
      Arguments:
         state:           "start"
         trigger_name     "attr_out"
         total_page_count "0"
         page_count       "0"
         proc_count       "0"
         total_proc_count "0"
         class            "body"
         just             "1"

    /*-------------------------------------------------------
    /*-- the meta event is for meta data on the doc
    /*-------------------------------------------------------

    Event Name is: doc_meta
        Arguments:
           state:           "start"
           empty:           "1"
           trigger_name     "attr_out"
           total_page_count "0"
           page_count       "0"
           proc_count       "0"
           total_proc_count "0"
           class            "body"
           just             "1"
```

Figure 4

In Figure 4 we see that the name of the event that was called is specified after the text “Event Name is:”. Underneath the event name we see name-value pairs in which the name of the event variable is on the left and its value is on the right. We also see no trace of the hockey data that was the subject of the PROC PRINT that we expect to be passed through ODS. That’s because the events above are among the first events called when only metadata is passed. Knowing that an event called DOC is called means that I can write a tagset that defines an event called DOC and any code defined by that event will be executed at least once. We also know that that code will be executed before the code associated with the DOC_HEAD event is executed. What we can’t tell from this output is if the DOC event will be called more than once. By looking at complete output files from tagsets such as EVENT_MAP and TEXT_MAP, we can start to detect patterns of event calling that are a function of the order in which data is sent to ouput.

We now know that output is written to markup files according to code found in event definitions, that events are called by SAS in an order that reflects the order it is found in the markup file that we can find out about through mapping tagsets, and that data is passed through these events through event variables, but we haven't yet seen how this facilitates the output of nested markup. The quick answer to this question is through states and patterns. As seen in the TEXT_MAP output above, one of the event variables is called STATE. Its two possible values are "start" and "finish". Many events are called by SAS in states. In other words, when SAS calls an event, it calls either its Start state or its Finish state. For that reason, we can also define an event (i.e. the code executed when the event is called) in states - the Start state which is the code executed when the Start state of the event is called, and the Finish state, the code executed when the Finish state of the event is called. Although there is often more to it, generally speaking, we use the Start state to generate the start tag of an element and the Finish state to generate its end tag.

After reviewing a few sample outputs from the mapping tagsets, it doesn't take long to realize that the sequence in which events and their states are called is no accident. Rather, it closely reflects the nested markup structure that is typical of tabular output (which of course most PROCs produce). Typically the list of events called when a tagset is used can be long (especially when the tagset, through an attribute statement, allows for stylesheets to be embedded into the file). Not pictured in the above output is the last event - the Finish state of the DOC event. The fact that DOC's Start state is the first event and its Finish state is its last reflects the fact that most markup structures require a root element. In other words, the two states of the DOC event are often used to generate the start and end tags of the root element. Later in the data transfer process, when a PROC produces a table, you'll start to see events called in a pattern that reflects tabular markup - one TABLE event for the table, one instance of a ROW event for each row in the table (data rows as well as header rows), and one DATA event for each cell of the table. If a table has ten data rows, one header row, and three columns (without row headers), then the Start state of a ROW event will be called, then three instances of a DATA event, then the Finish state of the ROW event. This pattern will repeat ten more times, once for each remaining row of the table. When defining a tagset to write HTML, you might define the Start state of the ROW event to write "<tr>" and the Finish state to write "</tr>". The DATA event doesn't need to be defined in states because its Finish state is called immediately after its Start state anyway. Knowing that the event variable VALUE carries data produced by the PROC, this event might include code such as the following:

```
put "<td> value </td>" ;
```

This has been a brief introduction to the world of tagsets and how they can be used to customize your markup structure. More information can be found in the SAS Online Documentation, but getting comfortable with tagsets also requires lots of practice and experimentation, in order to know what events are called and what data is available when. With the relatively complex structure of ODM and DEFINE-XML, writing a tagset may be a viable solution. The way that a tagset is written as well as the way it is called depends on how you manage and store your metadata. In my paper A SAS Programmer's Guide to Generating Define.xml (SAS Global Forum, 2009), I discuss a tagset I wrote that depends on a certain database structure used to store metadata. In the end, a PROC PRINT of each data set in the database in the order in which the metadata should appear in the Define.xml, wrapped in an ODS sandwich that specifies the use of this tagset, is what produces the desired output.

We now have a solution for exporting custom XML that frees us from the constraints of the out-of-the-box options that SAS gives us. On the import side, we now need something that will read XML of any structure rather than the structure dictated by, say, values of the XMLTYPE= option on the LIBNAME statement.

XML MAPS

While tagsets can be used for exporting custom XML markup, an XML *map file* can be used to import custom XML. A map file is a file that tells SAS how to navigate an XML file to determine which sections of the XML document determine data sets, observations, and variables, as well as where to get data to create a data set. XMLMap is another example of an XML schema, so a map file is made of well-formed XML. By pointing to this file on the LIBNAME statement, we can force SAS to interpret the XML file according to the instructions laid out in the map.

The root element of a map file is the `SXLEMAP` element. Underneath the root element is one or more instances of a `TABLE` element. Each `TABLE` element corresponds to a definition of a new data set. Of course to define a data set, we need to give it a name, some variables, and optionally, a label. The name of the table is provided in the name attribute of the `TABLE` element. A label can be provided as the value of the `TABLE-DESCRIPTION` child element. The name of each variable is provided in the name attribute of the `COLUMN` child element, of which there is one for each variable in the data set. In addition to these basic attributes of a data set, when information about observation boundaries and data values is to be extracted from an XML document, then the location of such information must also be part of the definition.

The `TABLE-PATH` element is a required child element of `TABLE` that defines observation boundaries. The value of this element is a path into the XML document that resembles Unix notation of directory paths, where directory names are replaced by element names. Such paths are specified with a syntax called XPath. XPath is a rich syntax with path reference shortcuts that take advantage of notions of current and relative paths, but for now, we'll keep our

discussion of XPath limited only to what we need. A path that begins with a forward slash (/) is an absolute path that begins at the root element. The path in the value of this element tells SAS that when reading the XML document, to define an observation between the start and end tags of this path. For example, the following tells SAS to start a new observation when it encounters the start tag of a `team` element, and to end that same observation when it encounters the corresponding end tag.

```
<TABLE-PATH>/nhl/team</TABLE-PATH>
```

Also underneath a `TABLE` element is one instance of a `COLUMN` element for each variable to be defined for the data set. `COLUMN` contains child elements such as `TYPE`, `DATATYPE`, `LENGTH`, `DESCRIPTION`, and `FORMAT`, each of whose values contain the obvious metadata for that variable. `COLUMN` also has a `PATH` element whose value is an XPath expression to the value of the variable. When the value is just a path, then the value of the element is extracted and passed on as the value of the variable. In the following example, the value of the variable character CONFERENCE is extracted from the value of the `Conference` element.

```
<COLUMN name=″conference″>
  <PATH>/nhl/team/Conference</PATH>
  <TYPE>character</TYPE>
  <LENGTH>15</LENGTH>
</COLUMN>
```

Attribute values can also be captured using the at sign (@). In the following example, the value of the `name` element is captured to create the data set variable NAME.

```
<COLUMN name=″name″>
  <PATH>/nhl/team/@name</PATH>
  <TYPE>character</TYPE>
  <LENGTH>15</LENGTH>
</COLUMN>
```

Other XMLMap functionality includes variable value retention and counter variables. Retention works the same way it does with the RETAIN statement in the DATA step. By default, when SAS begins to create a new observation, it sets all variable values to null, except those that are retained. A variable is retained by setting the value of the `retain` attribute on the `COLUMN` element to “YES”. This is necessary when the value of the `PATH` element for the variable being defined is at higher level than the observation boundary. XMLMap also allows for the definition of a counter variable whose value, rather than being extracted from the XML document, is incremented (or decremented) each time the path in the `INCREMENT-PATH` element value (a child of `COLUMN`) is encountered. For such variables, the `PATH` child element is not included, and the value of the `class` attribute of the `COLUMN` element is set to “ORDINAL”.

As an example, let’s suppose we want to read a Define.xml file, and create a data set to hold all of the controlled terminology we find. We’ll call this data set CODELISTITEMS and give it a label of “Define.xml CodeListItems.” This data set will contain one observation per codelist item per codelist. In this data set we’ll have the following variables: CTLIST_DESCRIPTION (description of the codelist), NCI_CL_CODE (NCI codelist level C-code), CODEDVALUE (the actual item or allowable value), DECODE (explanation of CODEDVALUE), and NCI_ITEM_CODE (NCI item-level C-code). The example below is a reminder of one way in which controlled terminology is stored in define.xml (omitting data in elements above the `CodeList` element).

```
<ODM>
    <Study>
    <MetaDataVersion>
    <CodeList OID="CL.AESEV" Name="Severity/Intensity Scale for Adverse Events"
    DataType="text" SASFormatName="$AESEV">
        <CodeListItem CodedValue="MILD" Rank="1">
         <Decode>
          <TranslatedText xml:lang="en">Grade 1</TranslatedText>
         </Decode>
         <Alias Name="C41338" Context="nci:ExtCodeID"/>
        </CodeListItem>
        <CodeListItem CodedValue="MODERATE" Rank="2">
          <Decode>
            <TranslatedText xml:lang="en">Grade 2</TranslatedText>
          </Decode>
          <Alias Name="C41339" Context="nci:ExtCodeID"/>
        </CodeListItem>
        <CodeListItem CodedValue="SEVERE" Rank="3">
          <Decode>
```

```
            <TranslatedText xml:lang="en">Grade 3</TranslatedText>
          </Decode>
          <Alias Name="C41340" Context="nci:ExtCodeID"/>
        </CodeListItem>
        <Alias Name="C66769" Context="nci:ExtCodeID"/>
      </CodeList>
```

Because we want our observation boundaries to be determined at each instance of the CodeListItem element, we start our mapping file in the following manner.

```
<?xml version="1.0" encoding="UTF-8"?>
<SXLEMAP version="1.2">

<TABLE name="CodeListItems">
 <TABLE-DESCRIPTION>Define.xml CodeListItems</TABLE-DESCRIPTION>
  <TABLE-PATH
syntax="XPath">/ODM/Study/MetaDataVersion/CodeList/CodeListItem</TABLE-PATH>
```

This ensures that a new observation will be created every time a new `CodeListItem` element is read.

The value of CTLIST_DESCRIPTION can be extracted from the Name attribute of the `CodeList` element.

```
<COLUMN name="CTLIST_DESCRIPTION" retain="YES">
        <PATH syntax="XPath">/ODM/Study/MetaDataVersion/CodeList/@Name</PATH>
        <TYPE>character</TYPE>
        <DATATYPE>string</DATATYPE>
        <LENGTH>200</LENGTH>
        </COLUMN>
```

Here we note that the `retain` attribute is set to “YES”. This way, since a new observation is started at each new instance of `CodeListItem`, this value is carried along into the next observation. The same is true with other variables that are captured above the level of the observation boundary, as in NCI_CL_CODE below, which is captured in the `Name` attribute of the Alias child element of `CodeList`.

```
<COLUMN name="NCI_CL_CODE" retain="YES">
          <PATH>/ODM/Study/MetaDataVersion/CodeList/Alias/@Name</PATH>
          <TYPE>character</TYPE>
          <LENGTH>10</LENGTH>
        </COLUMN>
```

The rest of the variables can be captured from attribute and element values at or below the level of the observation boundary, and so do not need to be retained. The following completes the map by extracting the rest of the variables.

```
<COLUMN name="CODEDVALUE">
 <PATH
syntax="XPath">/ODM/Study/MetaDataVersion/CodeList/CodeListItem/@CodedValue</PATH>
  <TYPE>character</TYPE>
  <DATATYPE>string</DATATYPE>
  <LENGTH>200</LENGTH>
</COLUMN>

<COLUMN name="DECODE">
<PATH
syntax="XPath">/ODM/Study/MetaDataVersion/CodeList/CodeListItem/Decode/TranslatedTe
xt</PATH>
  <TYPE>character</TYPE>
  <DATATYPE>string</DATATYPE>
  <LENGTH>200</LENGTH>
</COLUMN>

<COLUMN name="NCI_ITEM_CODE">
    <PATH>/ODM/Study/MetaDataVersion/CodeList/CodeListItem/Alias/@Name</PATH>
    <TYPE>character</TYPE>
    <LENGTH>10</LENGTH>
</COLUMN>
</TABLE>
</SXLEMAP>
```

In Version 2.0 of DEFINE-XML, controlled terminology can be found not only in `CodeListItem` elements, but also in `EnumeratedItem` elements (child elements of CodeList). `EnumeratedItem` is used for codelist items that don't need a decode.

```
<CodeList OID="CL.ACN" Name="Action Taken with Study Treatment" DataType="text">
  <EnumeratedItem CodedValue="DOSE NOT CHANGED" OrderNumber="1">
    <Alias Name="C49504" Context="nci:ExtCodeID"/>
  </EnumeratedItem>
  <EnumeratedItem CodedValue="DOSE REDUCED" OrderNumber="2">
    <Alias Name="C49505" Context="nci:ExtCodeID"/>
  </EnumeratedItem>
  <EnumeratedItem CodedValue="DRUG INTERRUPTED" OrderNumber="3">
    <Alias Name="C49501" Context="nci:ExtCodeID"/>
  </EnumeratedItem>
  <EnumeratedItem CodedValue="DRUG WITHDRAWN" OrderNumber="4">
    <Alias Name="C49502" Context="nci:ExtCodeID"/>
  </EnumeratedItem>
  <Alias Name="C66767" Context="nci:ExtCodeID"/>
</CodeList>
```

Because the observation boundary is different, this controlled terminology will need to be extracted to a different data set. This map won't be much different from the one above and can be accomplished in the same map file under a new `TABLE` element. This exercise is left for the reader.

With the map file now in place, all that's left is to point SAS in the right direction. We first establish a fileref to the map file, and then use that fileref as the value of the XMLMAP= option on the LIBNAME statement.

```
filename mymap 'C:\mymap.map' ;
libname xmlfile xml 'C:\myxml.xml' xmlmap=mymap ;
```

XMLMap is another technology that, while much quicker to learn than tagsets, is still something else for the SAS programmer to add to their toolbox. The good news is that SAS offers the XML Mapper - a point-and-click interface that allows you to create a map without having to know anything about the XMLMap structure or about XPath. XML Mapper allows you to open an XML document and design a SAS data set (or more than one) by dragging and dropping elements and attributes from the XML file. While you do this, an XML map is generated that you can review and save.

CONCLUSION

XML is becoming more and more significant as standards are further developed within the CDISC community, and most SAS programmers in the industry are going to have to work with data in this format at some point. We may not need to become experts on the subject but we do need to at least become aware of what XML is, what the rules are surrounding its structure, how it's used in different contexts. SAS has come up with a number of tools that range from restrictive, but easy to use, to flexible but with a learning curve. Even if it hasn't yet come up in your workplace, there's a good chance it's coming. It's never too early to start experimenting.

REFERENCES

Molter, Michael J. 2009. "A SAS Programmer's Guide to Generating Define.xml." *Proceedings of the SAS Global 2009 Conference*. Available at http://support.sas.com/resources/papers/proceedings09/163-2009.pdf.

CONTACT INFORMATION

Your comments and questions are valued and encouraged. Contact the author at:

Mike Molter
d-Wise
1500 Perimeter Park Drive, Suite 150
Morrisville, NC 27560
919-600-6237
mike.molter@d-wise.com
www.d-wise.com