Using SAS® and Other XML Tools Effectively

Scott E. Chapal, JWJERC Ichauway Inc., Newton, GA

Abstract

SAS programmers who have XML processing needs, face a bewildering array of tools and standards to consider. The XML tools in SAS include ODS MARKUP, PROC TEMPLATE, and the LIBNAME XML engine with XMLMap. However, particular applications might be enhanced by the use of other tools and approaches also.

There are many other technologies available for XML document modeling, processing and presentation. This paper attempts to sort through some of the standards-based XML tools to provide a view from a SAS-centric perspective. The utility of these XML supporting technologies will be considered in relation to the functionality of the SAS XML tools. Emphasis will focus on Java-based examples.

Introduction

In the past few years, the eXtensible Markup Language (XML) has become the prevailing standard markup language used to provide structured information. Due to it's configurability and open format, XML has even been termed “Self Describing Data”. Although often over-hyped, XML *has* penetrated Information Technology to such an extent that even we SAS programmers have no choice but to try to understand the impact of XML and the processing tools available to us.

A cursory assessment might lead one to conclude that XML is just another markup language and that deciding how to create and use XML is, well, simple. This view may be tied to the success of HTML, ironically, since the markup provided by HTML is presentation focused whereas XML provides the capacity for markup based on the content of the data. Some XML problems *are* simple, but there are many purposes for XML and many processing options and the choices among those options are sometimes not obvious. The XML supporting technologies and associated standards (XMLSchema, XSLT, XPath, etc.) as well the proliferation of XML vocabularies which are maturing rapidly (DocBook, ebXML CDISC, for example), make it challenging to understand the XML landscape. Additionally, there are numerous XML software options, both commercial and open source which also complicate the decision making. So, a walk through some of the prominent standards and tools might help to clarify how to use this XML toolkit more effectively in a SAS environment.

Standards considerations

Since XML was derived from SGML (Standard Generalized Markup Language - ISO 8879:1986), the structure of an XML document was defined in a DTD (Document Type Definition) until recently. Now, however, other means of defining document structure have been developed, notably XML Schema which was approved as a W3C Recommendation in 2001.

XMLSchema is an XML language for defining and constraining the content of XML documents including those which exploit the XML Namespace facility. XMLSchema validation can be extended by additional constraints expressed in XSLT/XPath to supplement limitations in XMLSchema. “The purpose of a schema is to define a class of XML documents, and so the term ”instance document” is often used to describe an XML document that conforms to a particular schema.”(*XML Schema Part 0: Primer* 2001) A validating parser can be used to verify the structure of an instance document against the constraint rules provided in the XML Schema. XML Schemas have certain advantages over DTD's. An XML Schema can define datatypes and other complex structures that are difficult or impossible to do in a DTD. Schemas are themselves XML documents, so XSLT stylesheets can be written to manipulate them.

XML Namespaces provides an extensibility mechanism for the modular re-use of XML vocabularies (*Namespaces in XML* 1999). An XML Namespace, identified by a URI in an XML document, provides an unambiguous reference to a markup vocabulary which is defined elsewhere. Thus, XML documents can be assembled from component “namespaces” with the ability to distinguish between duplicate element type and attribute names.

XSLT Extensible Stylesheet Language Transformations (Clark 1999) is a W3C standard designed for manipulating XML to other formats: text, HTML, other XML formats, or even PDF. XSLT uses a template based approach to addressing and transforming selected parts of an XML document via XPath notation(Clark & DeRose 1999). “Generally, XSLT provides a series of operations and manipulators, while XPath provides precision of selection and addressing”(Gardner & Rendon 2002). XSLT/XPath is a core XML technology which is well supported and well understood and is likely to be around for a long time.

XSLT has been criticized for being a difficult language to learn and use, however it *is* ideally suited to treat an XML document as a complex data structure by providing powerful tools for extracting that information into derivative data representations. The effort involved in creating XSLT stylesheets is not wasted if other components of a system change over time (ie. Java or even SAS!). Core XSLT capabilities include: string manipulation, numerical operations, Dates and Times and XML Transformation. XSLT capabilities can also be enhanced through extensibility to use functions in code defined external to the XSLT processor. XSLT 2.0 and XPath 2.0 are being developed in conjunction with one another.

Rationale for XML

For SAS programmers and data managers, understanding the XML capabilities provided by SAS in relation to other XML technologies is important when planning for applications. A motivation for using XML in the first place is to achieve flexible, extensible markup. Some design choices can restrict the flexibility of XML. Design considerations for SAS and XML integration might include planning for content markup, data exchange and document modeling.

Content Markup

One of the design objectives of XML applications is the deliberate separation of content from presentation. This achieves re-usability of content decoupled from considerations of presentation. A similar concept is contained in the Model-View-Controller (MVC) design paradigm used to develop interactive web-tier applications. The Model, View and Controller aspects are intentionally separate in their respective functions, but in combination form a cohesive, flexible system. An example is the Apache Project **Struts** web application framework which supports the MVC (Model 2) design pattern by providing a controller component which integrates with other model and view components. The benefits of MVC include encouraging code reuse, centralizing control and ease of modification.

Likewise, the separation of content from presentation, possible with XML, provides a division of functionality which makes design and maintenance clearer. XML provides for extensible markup vocabularies where the structure, meaning and relationships of the data can be designed to be self-evident. Well-designed XML markup is internally consistent (well-formed and valid) and appropriate to a particular purpose. Unlike HTML, where data and formatting instructions are intertwined, XML can provide structured content-markup. Thus, XML used in combination with validation criteria (Schema) supports the goal of separating content from presentation in applications.

Data Exchange

Exchanging data within and between applications is an ascendant role for XML due to its platform independence, flexible expressivity and support for Unicode. SAS itself is incorporating Unicode support starting in 9.1(Beatrous 2003). Further, XML data can function to *decouple* systems making the components easier to maintain and replace. This *loose-coupling* is one of the precepts of **Web Services** which promises the extension of complex services via XML over the Internet. Web Services, sometimes characterized as a mere retread of the RPC concept, has a high potential for adoption in large part because XML standards provide a a common framework which serves to stabilize and orient development.

XML is widely used for data exchange in the form of “Messaging”. A message simply defined as a collection of data fields organized into a *header* and a *payload*, which is a structure that is easily expressed in XML. There are relevant API’s, such as the the Java API for XML Messaging (JAXM), and standards, including the Simple Object Access Protocol (SOAP) for messaging and RPC. When an application necessitates “loose coupling” of the components, (as in the evolving “Web Service” model), XML messaging provides a flexible structured data transport medium.

ebXML (electronic business using eXtensible Markup Language) “is a modular suite of specifications that enables enterprises of any size and in any geographical location to conduct business over the Internet. Using ebXML, companies now have a standard method to exchange business messages, conduct trading relationships, communicate data in common terms and define and register business processes” `www.ebxml.org`.

CDISC is an example of a standards driven, domain-specific XML vocabulary gaining broad use. The Clinical Data Interchange Standards Consortium is an open, multidisciplinary, non-profit organization. CDISC is committed to the development of worldwide industry standards to support the electronic acquisition, exchange, submission and archiving of clinical trials data and metadata for medical and bio-pharmaceutical product development. A goal of CDISC was to base the Operational Data Model on XML technology with eventual support for XML Schemas. The CDISC XML example is also relevant due to the prominence of SAS in clinical data management and the historical use of the v5 transport format for submissions.

Document Modeling

Metadata is often defined as data about data. Using this most basic definition, it could be said that XML *is* (or at least contains) meta-data! Beyond the simplistic definition however, the term meta-data often implies a specific highly structured representation of a data resource or application. This kind of highly structured metadata is another role to which XML is well suited. For instance, the Dublin Core Metadata Initiative provides a set of standardized elements for tagging metadata. “The Dublin Core metadata element set is a standard for cross-domain information resource description.”(DCMI 2003). Additionally, relationships could then be assembled using XMLSchema (or RDF - Resource Description Framework), to combine elements from distinct namespaces into modular metadata schemas. Although those details are beyond the scope of this paper, the point is that XML *can* be used to build complex metadata which is human readable, web accessible and machine process-able. Those modular metadata schema are an example of the sophisticated document modeling jobs to which XML is suited.

DocBook is an example of a mature SGML/XML document modeling vocabulary. “DocBook is a very popular set of tags for describing books, articles, and other prose documents, particularly technical documentation”(Walsh & Muellner 2002). DocBook developed as an SGML DTD which provides a large and robust definition for writing structured documentation (books) with a growing community of authors. It is relevant to this discussion because: 1) DocBook provides a standard markup vocabulary for technical documentation including software and 2) ODS MARKUP provides a Docbook tagset. There is an experimental version of DocBook 4.1.2 available as an XMLSchema[1], possibly presaging an official release.

XML from a SAS Perspective

Understanding the XML capabilities in SAS relative to other (non-sas) XML tools is essential to make more effective use of these tools independently or in conjunction with one another.

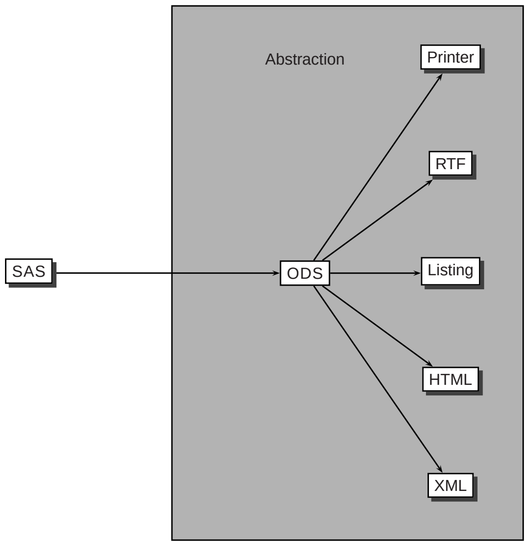
SAS XML Facilities

The XML features in SAS are in some ways “interdependent” but they also seem to form a layered system of functionality. ODS provides destinations for generating flexible output and also supplies the foundation for tagsets available to LIBNAME XML through ODS MARKUP and PROC TEMPLATE.

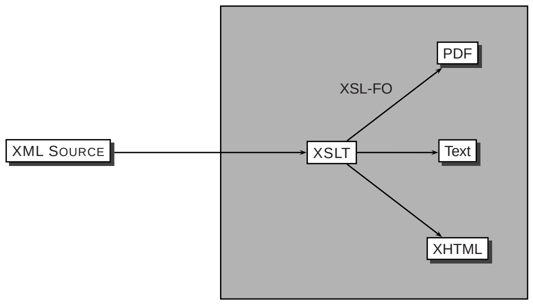
ODS

The Output Delivery System has been in development since *before* the full impact of XML was well established. In fact, XML tagsets were not incorporated into SAS until relatively recently and most of the ODS output destinations are *presentation* markup (Table 1). The “abstraction”(Figure 1a) of output types provided by ODS consists of a collection of ’destinations’ primarily designed for formatting SAS-generated output. Much of the utility of ODS is its power to generate formatted tabular reports, however there are graphical and page layout capabilities also. It is also the case that many of these output formats are now possible using XML as the common format and tools such as XSLT(Figure 1b).

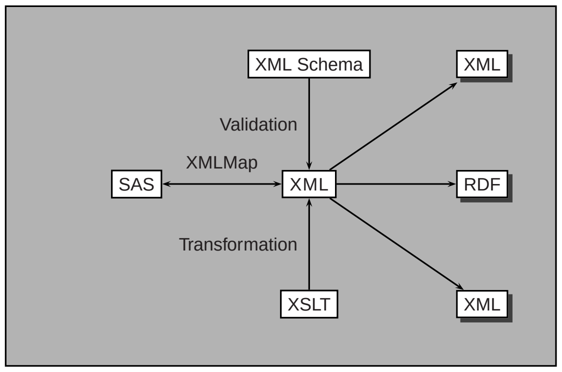
Beginning with version 9.0, SAS provides a new formatted ODS destination called ODS DOCUMENT. An ODS document is a “hierarchical file of output objects that is



(a) ODS generates output tables to multiple destinations including XML via ODS MARKUP.



(b) XML transformation via XSLT to multiple formats achieves similar results. XSL-FO provides Formatting Objects into PDF.



(c) XML can be validated via XML Schema and transformed via XSLT to accomplish various, or multiple document requirements.

Figure 1: The Output Delivery System and XML/XSL transformation tools provide distinct *and* overlapping functionality.

[1]http://www.oasis-open.org/docbook/xmlschema/4.1.2.3/

ODS Destination	Purpose	Version Introduced
LISTING	Default (output)	8[a]
OUTPUT	Output Data Set	8
PRINTER	PS, PCL, PDF	8
HTML	HTML 3.2	8
RTF	RTF	8.2
DOCUMENT	"Raw" output stream	9.0
MARKUP	Tagsets	9.0[b]

[a] v8 destinations originated in v7?
[b] Pre-production 8.2

Table 1: The purpose of ODS destinations has been expanded to include the DOCUMENT destination as well as the flexible MARKUP tagset facility.

created from a procedure or data query". The ODS document resides in a SAS library and and has the persistence characteristics of other library members. ODS Document would seem to accomplish "content markup" in much the same way that an XML vocabulary could, although ODS documents are not portable across operating systems, presumably.

Hierarchical ⇔ Rectangular Transformation

Getting XML data into and out of data sets is a basic requirement for SAS applications. The problem of taking XML encoded data and translating it into a record-oriented SAS data set structure is straight forward for "regular"[2] XML and non-trivial for "hierarchical" XML forms. In reality however, a lot of XML markup is document-oriented (very hierarchical) and may contain additional complexities of IDs, Links, sub-documents and so on. Obviously, the hierarchical nature of XML data representation contrasts distinctly with the 'rectangular' nature of SAS data sets and illustrates the need for efficient and automate-able means of 'transformation'.

Libname XML

The SAS XML Libname Engine (SXLE) provides an import/export method for inter-converting SAS data sets to 'regular' XML. The export of XML[3] uses familiar libname syntax:

```
libname export xml 'class.xml';
data export.class;
    set sashelp.class;
    run;
```

The resulting XML is 'regular' with the document root element `<TABLE/>` containing a series of `<DATASET/>` elements, (corresponding to the data set name) each of which contains elements corresponding to the variables in the data set: Name, Sex, Age etc.

[2] Essentially, XML markup which *already* represents a rectangular, or minimally hierarchical (Table 1) organization, of the data!

[3] v9.0 default uses the SASXMOG (Table 2) Tagset

```
<?xml version="1.0" encoding="iso-8859-1" ?>
<TABLE>
   <CLASS>
      <Name> Alfred </Name>
      <Sex> M </Sex>
      <Age> 14 </Age>
      <Height> 69 </Height>
      <Weight> 112.5 </Weight>
   </CLASS>
```

Importing data via Libname XML is similar:

```
libname import xml 'regular_class.xml';
data class;
    set import.class;
    run;
```

and requires a specifically structured XML document, generally of the form:

```
<?xml version="1.0" ?>
<DATA>
   <DATASET1>
      <VARIABLE1> Variable1_Value </VARIABLE1>
      <VARIABLE2> Variable2_Value </VARIABLE2>
      ...
   </DATASET1>
....
   <DATASET2>
      ...
   </DATASET2>
...
</DATA>
```

The document-root element `<DATA/>`, contains repeating instances of elements named for the data set they represent `<DATASET1/>, <DATASET2/>`, each of which contain the variables for the data set.[4]

XMLMap

The 'regular' XML requirement in libname XML feature can be overcome with XMLMap to import and export 'non-regular' XML. XMLMap utilizes XPath (Clark & DeRose 1999) notation and version 9.1 will use dataTypes which conform to XML Schema DataTypes specification. An XMLMap fragment shows the hierarchical structure:

```
  <TABLE name="Class">
   <TABLE-PATH syntax="xpath">
     /class:Class/Grade/Group/Student
     </TABLE-PATH>
    <COLUMN name="Age" retain="yes">
     <PATH>/class:Class/Grade/@Age
```

[4] Other XML formats can be generated/invoked using the XMLTYPE= option. Supported output types in v9.0 are: GENERIC, ORACLE, OIMDBM, EXPORT, HTML (export is currently aliased to OIMDBM). The XMLMETA= option to libname XML provides for "metadata-related" information in the markup, most obvious in the OIMDBM format. However, as of 9.0 the OIM XMLTYPE has been deprecated and support for OIM is being discontinued (Friebel 2003), with the functionality being shifted to other XMLtypes.

```
     </PATH>
     <TYPE>character</TYPE>
     <DATATYPE>INTEGER</DATATYPE>
     <LENGTH>3</LENGTH>
    </COLUMN>
    <COLUMN name="Sex" retain="yes">
     <PATH>/class:Class/Grade/Group/@Sex
```

...
A simple invocation will read the MAPped data.

```
filename in './output/class_custom_put.xml';
filename map './class_map.xml';
libname  in xml xmlmap=MAP;

proc print data=in.Class; run;
```

Building and debugging the XMLMap can be challenging, as SAS does not seem to provide very diagnostic error messages. The XML Mapper (Atlas) application is a graphical interface used to assist in XMLMap creation, and is reported to support XML Schema in 9.1. XSLT might be a viable alternative or companion technology for handling convoluted XML data structures.

Using Schema

Recall the `CLASS` data rendered by Libname XML:

```
<?xml version="1.0" encoding="iso-8859-1" ?>
<TABLE>
   <CLASS>
      <Name> Alfred </Name>
      <Sex> M </Sex>
      <Age> 14 </Age>
      <Height> 69 </Height>
      <Weight> 112.5 </Weight>
   </CLASS>
```

...
A description of this class of document(s) could be declared in a simple XML Schema:

```
<?xml version="1.0" encoding="UTF-8"?>
<schema xmlns="http://www.w3.org/2001/XMLSchema">
 <element name="TABLE"
          minOccurs="1" maxOccurs="1">
  <complexType>
   <sequence>
    <element name="CLASS"
             minOccurs="1" maxOccurs="unbounded">
    <complexType>
     <sequence>
      <element name="Name" type="string"/>
      <element name="Sex" type="string"/>
      <element name="Age" type="integer"/>
      <element name="Height" type="decimal"/>
      <element name="Weight" type="decimal"/>
     </sequence>
    </complexType>
    </element>
   </sequence>
  </complexType>
 </element>
</schema>
```

Notice that the structure of the schema is explicit and is more-or-less obvious. `<CLASS/>` (a complex element) must occur one or more times and is composed of `Name, Sex, Age, Height and Weight` elements, each declared with a respective type[5]. Element sequence, repeatability, optionality and associated attributes are all definable in the Schema (however this topic is far more involved than the brief treatment given here).

In the previous instance document example, Name, Sex, Age, Height and Weight are all component elements of CLASS. Consider a different XML document requirement which is more 'hierarchical' than the libname XML default. You could distinguish the `Name` of the Student as a distinct subject, having the measurable properties of height and weight, and belonging to a sex group and of a certain age. In this example, suppose that the quantities of interest in the class data set are height and weight and the other variables *could* be considered as groups or `meta-data`... so their values might well be portrayed as attributes in the instance document.

```
  <Group Sex="F">
   <Student Name="Joyce">
```

Perhaps a more “Russian-doll” representation of these data might be appropriate for some purpose:

```
 <Grade Age="11">
  <Group Sex="F">
   <Student Name="Joyce">
     <Height> 51.3 </Height>
     <Weight> 50.5 </Weight>
   </Student>
  </Group>
  <Group Sex="M">
   <Student Name="Thomas">
     <Height> 57.5 </Height>
     <Weight> 85 </Weight>
   </Student>
  </Group>
 </Grade>
 <Grade Age="12">
```

...
If, in fact, height and weight are of quantitative interest, it might be important to include a definition of the units of measurement. These could be included as attributes, for example:

```
<Height unit="inch">
<Weight unit="pound">
```

and repeated for every `<Student/>`. But they might also be declared once, if their use was consistent in the instance document:

[5] See `http://www.w3.org/TR/xmlschema-2/` for a discussion of XML Schema DataTypes.

```
<units>
  <ageunit name="year"/>
  <htunit name="inch"/>
  <wtunit name="pound"/>
</units>
```

A data step approach to generate this format is shown in Appendix I and a somewhat more involved XML Schema which validates this format is outlined in Appendix II.

Having the ability generate an XML Schema for content created by SXLE would be important for automating XML based applications. A new tagset `sasxmxsd` has been described for Version 9 (Friebel 2003) to provide this functionality:

```
libname xsdgen xml 'xml-with-xsd.xml'
        tagset=tagsets.sasxmxsd;
```

Using XML Tagsets

As mentioned previously, XML output *can* be generated in a SAS data step with put statements:

```
put '<doc-root>';
put '  <element>' value '</element>';
...
put '</doc-root>';
```

But, it is obvious that this is tedious, potentially error prone (well-formedness is not even checked), and there are better ways.

The SAS supplied tagsets are a collection of markup styles ranging from TROFF to XML (Table 2). The XML tagset names that start with “SAS” are used with the libname XML engine for reading and writing XML encoded data. Using the ODS MARKUP statement without an explicit tagset declaration, creates an XML document. The basic invocation uses a file (and path, if not the current directory) descriptor to write to. The ODS XML destination is an alias to invoke this default XML output.

```
ods markup path='output'
           file='default.xml';
proc whatever;
   ...;
ods markup close;
```

The ID= option is useful to route output to multiple instances of the same destination (in this case XML), each with different options (type = is an alias for tagset=).

```
ods markup (id=1) tagset=default
    file  = "default.xml"
ods markup (id=2) type=event_map
    file="default_event_map.xml";
```

File-type specifications can be used with ODS MARKUP XML tagsets to generate other XML specific files. Using type=default produces an xsl file, a DTD and a Cascading stylesheet.

```
ods markup
    type  = default
    code  = "default.xsl"
    frame = "default.dtd"
    stylesheet = "default.css"
```

The output from these options appears to have changed substantially from SAS 8.2 to 9.0. The XSLT file produced by code= generates instructions for HTML transformation, as can be seen in the xsl:output element of the generated XSL file:

```
<xsl:output method="html"/>
```

By using the code= and stylesheet= file-specifications together with the default tagset, the XSLT file references the Cascading Stylesheet, as seen in this `default.xsl` fragment:

```
<?xml version="1.0"?>
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns="http://www.w3.org/TR/REC-html40">
<xsl:output method="html"/>
<xsl:template match="odsxml">
  <HTML>
    <HEAD>
      <TITLE><xsl:value-of select="proc/title"/></TITLE>
      <LINK REL="STYLESHEET" HREF="class.css">
    ...
```

The above fragment is also useful for introducing the concept of style inheritance for custom tagset generation with PROC TEMPLATE. There is a problem in the LINK element within the `<xsl:template match="odsxml">` rule as shown above. Even though the stylesheet is *creating* HTML, the stylesheet itself needs to be well-formed XML. (HTML doesn’t care, XML does). The LINK element has no closing LINK tag, or a trailing / before the closing angle bracket. So the element should really look like:

```
  <LINK REL="STYLESHEET" HREF="class.css"/>
```

Inheritance can be used to replicate a tagset with certain events (re)defined if necessary to apply changes or enhancements to the tagset:

```
    define event code_stylesheet_link;
      putq "      <LINK REL=""STYLESHEET"" HREF=" url "/>"
          NL / if exist( url );
   end;
```

See Appendix V for a program which corrects the tag and applies it.

Proc Template

Version 9.0 uses the SASXMOG Tagset by default for writing XML via libname XML. It’s instructive to look at the tagset definition from PROC TEMPLATE by using the SOURCE statement to decompile the definition:

```
proc template;
   source tagsets.sasxmog;
```

TAGSET	Markup
	HTML
CHTML	CHTML
HTML4	HTML 4 w/ Stylesheet
HTMLCSS	HTML w/ CSS
IMODE	IMode
PHTML	Plain HTML
WML	WML
WMLOLIST	WML
	$\LaTeX$
COLORLATEX	Color $\LaTeX$
LATEX	$\LaTeX$
	CSV
CSV	Tabular CSV
CSVALL	CSV
CSVBYLINE	CSV
	XML
DEFAULT	ODSXML (Generic XML)
DOCBOOK	OASIS DocBook
PYX	Pyxie
	Events
EVENT_MAP	Events
NAMEDHTML	Diagnostic
SHORTMAP	Events (Short)
TEXTMAP	Events (Text)
TPL_STYLE_LIST	Events (HTML)
TPL_STYLE_MAP	Events (XML)
	Graphics
GTABLEAPPLET	GTableApplet
	LIBNAME XML
SASIOXML	Generic XML
SASXMOG	Oracle8iXML Generic[a]
SASXMOH	Simple HTML
SASXMOIM	OIM XML
SASXMOR	Oracle8iXML
	Printer
TROFF	Troff

[a] LIBNAME XMLTYPE=Generic

Table 2: ODS MARKUP tagsets documented in SAS v9.0

This reveals that `SASXMOG` is derived from:

```
   parent = tagsets.sasioXML;
```

Likewise `SASIOXML` shows:

```
   parent = tagsets.sasXML;
```

And finally `SASXML` consists only of:

```
define tagset Tagsets.Sasxml;
   notes "SAS XML model";
   define event XMLversion;
      put "<?xml version=""1.0""";
      putq " encoding=" ENCODING;
      put " ?>" NL;
      break;
   end;
   define event XMLcomment;
      put "<!-- " NL;
      put "      " TEXT NL;
      put "  -->" NL;
      break;
   end;
   mapsub = %nrstr("/&lt;/&gt;/&amp;/&quot;/&apos;/");
   map = %nrstr("<>&""'");
   indent = 3;
end;
```

which is just the XMLversion and XMLcomment events, that are common to all XML tagsets. Without digging into the details, this at least illustrates the inheritance model which is used to define SASXMOG: `SASXML` ⇒ `SASIOXML` ⇒`SASXMOG`

ODS diagnostic (events) tagsets can be used to understand event sequences and therefore to extend or design new tagsets (Gebhart 2002). Creating sophisticated custom tagsets requires familiarity with many aspects of PROC TEMPLATE, notably DEFINE EVENT, including the trigger and put* statements and conditional logic syntax. Also important is an understanding of the implications of the PROC TEMPLATE inheritance mechanisms for generating XML, including style-definition and style-element inheritance. There is a learning curve.

Other XML tools

Java implementations

Xalan The Apache project has been prolific in creating open source software, not the least of which is the Xalan XSLT processor. Xalan-Java can be used in command-line mode:

```
java org.apache.xalan.xslt.Process
    -XSL class.xsl -IN class.xml
```

This is useful for development, batch processing, or even invoking from SAS (Appendix V). More commonly, however, Xalan is probably used in a programmatic context. As a component of a servlet, for example, responding to requests for XML documents by transforming those documents into HTML and serving them to web browsers. Xalan supports the JAXP “pluggability” concept for transformation and parsing and supports creating and processing DOM trees.

JAXP The Java API for XML Processing (JAXP) enables applications to parse and transform XML documents using an API that is independent of a particular XML processor implementation. JAXP supports processing of XML documents using DOM, SAX and XSLT. It is essentially an abstraction which allows the substitution of different parser in an application without changing the application code. This achieves vendor/implementation independence of parsers: “It encapsulates differences between various XML parsers, allowing Java programmers to use a consistent API regardless of which parser they use”(Burke 2001).

JAXB The Java Architecture for XML Binding (JAXB) "provides an API and tools that automate the mapping between XML documents and Java objects" `java.sun.com/xml/jaxb/`. JAXB achieves access to XML data by compiling (`xjc`) an XML schema into Java classes. This provides fine-grained control of XML content into and out of Java objects including validation of the Java representation against schema constraints. Castor is another (open-source) Java-based framework which provides XML/Java/SQL binding, including object persistence to a RDBMS `castor.org`.

Parsing and validating is a ubiquitous requirement for XML processing, and the standard in the Java world seems to have settled on Xerces (`xml.apache.org`). Xerces is a fully conforming XML Schema parser which includes DOM and Namespaces processing capabilities. Another useful tool, especially when designing or debugging Schemas, is SUN's Multi-Schema Validator: [6] which provides a command-line tool for schema (DTD/RELAX/TREX/W3C) validation and supplies very informational error messages.

Conclusions

Figuring out how to combine the capabilities of SAS with other XML tools can be daunting. This whirlwind overview of some of the standards and XML technologies is a starting point, but certainly just that – a start. Many of the features of SAS 9.1 are not yet available or fully understood. As developments continue in SAS, including WebAF and Integration Technologies, they may provide new capabilities to address XML problems. It might be difficult to assess PROC TEMPLATE or XSLT, but the challenge, as always, is to choose wisely.

> There are boundaries, but they might be difficult to discern. – Chinese Fortune (Cookie)

Acknowledgments

I would like to thank Ian Whitlock and Ed Heaton for encouraging me to participate in SESUG 2003.

Contact Information

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

Scott Chapal
JWJERC
Ichauway, Inc.
Rt. 2 Box 2324
Newton GA. 39819
scott.chapal@jonesctr.org

[6] `http://wwws.sun.com/software/xml/developers/multischema/`

References

Beatrous, S. (2003). Multilingual computing with the 9.1 SAS unicode server, *Proceedings of the Twenty-Eighth Annual SAS® User Group Conference*, SAS Institute, Inc.

Burke, E. M. (2001). *Java and XSLT*, first edn, O'Reilly.

Clark, J. (1999). Xsl transformations(XSLT) version 1.0, "`http://www.w3.org/TR/xslt`".

Clark, J. & DeRose, S. (1999). XML path language (xpath) version 1.0, "`http://www.w3.org/TR/xpath`".

DCMI (2003). "`http://dublincore.org/documents/dces/`".

Friebel, A. (2003). XML? we do that!, *Proceedings of the Twenty-Eigth Annual SAS Users Group International Conference*, SAS Institute, Inc, pp. 173–28.

Gardner, J. R. & Rendon, Z. L. (2002). *XSLT & XPATH: A guide to XML transformations*, Prentice Hall.

Gebhart, E. (2002). Markup: The power of choice and change,, *Proceedings of the Twenty-Seventh Annual SAS® User Group Conference*, SAS Institute, Inc.

Namespaces in XML (1999). "`http://www.w3.org/TR/REC-xml-names/`".

Walsh, N. & Muellner, L. (2002). *DocBook The Definitive Guide*, 2nd edn, O'Reilly.

XML Schema Part 0: Primer (2001).

Appendices

Appendix I

SAS class data set custom XML via put statements.

```
filename custom './output/class_custom_put.xml';
data class;
    set sashelp.class;
proc sort data=class;
    by age sex name;
run;

data _null_;
  file custom;
  set class nobs=Last;
  by age sex name;
  if _n_=1 then do;
      put '<?xml version="1.0" ?>';
      put '<class:Class';
      put '  xmlns:class="http://class.org/Class">';
      put '<!-- Units Metadata -->';
      put '<units>';
      put '  <ageunit name="year"/>';
      put '  <htunit name="inch"/>';
      put '  <wtunit name="pound"/>';
      put '</units>';
      put '<!-- Class Data -->';
      output;
      end;
  if first.age then do;
      put ' <Grade Age="' age +(-1) '">';
      end;
  if first.sex then do;
      put '  <Group Sex="' sex +(-1) '">';
      end;
      put '   <Student Name="' name +(-1) '">';
      put '     <Height> ' height '</Height>';
      put '     <Weight> ' weight '</Weight>';
      put '   </Student>';
  if last.sex then do;
      put '  </Group>';
      end;
  if last.age then do;
      put ' </Grade>';
      end;
  if _n_ = Last then do;
      put '</class:Class>';
      output;
      end;
run;
```

Appendix II

XML Schema to validate the custom class XML.

```
<?xml version="1.0" encoding="UTF-8"?>
<schema
        xmlns="http://www.w3.org/2001/XMLSchema"
        xmlns:class="http://class.org/Class"
        targetNamespace="http://class.org/Class">
 <element name="Class"
          minOccurs="1" maxOccurs="1">
  <complexType>
   <sequence>
    <element name="units"
             minOccurs="1"
             maxOccurs="1">
     <complexType>
      <sequence>
      <element name="ageunit"
               minOccurs="1"
               maxOccurs="1">
      <complexType>
       <attribute name="name"
                  use="required"
                  default="year"/>
      </complexType>
      </element>
      <element name="htunit">
      <complexType><!-- Height units -->
       <attribute name="name"
                  use="required"
                  default="centimeter">
        <simpleType>
         <restriction base="string">
          <enumeration value="inch"/>
          <enumeration value="centimeter"/>
         </restriction>
        </simpleType>
       </attribute>
       </complexType>
      </element>
      <element name="wtunit">
      <complexType><!-- Weight units -->
       <attribute name="name"
                  use="required"
                  default="kilogram">
        <simpleType>
         <restriction base="string">
          <enumeration value="pound"/>
          <enumeration value="kilogram"/>
         </restriction>
        </simpleType>
       </attribute>
       </complexType>
      </element>
      </sequence>
     </complexType>
    </element><!-- Units -->
    <element name="Grade"
             minOccurs="1"
             maxOccurs="6"><!-- 6 yr range -->
     <complexType>
     <attribute name="Age"
                type="integer"
                use="required"/>
     <sequence>
       <element name="Group"
                minOccurs="1"
                maxOccurs="2"
                use="required"><!-- 2 Groups-->
       <complexType>
       <attribute name="Sex"
                  type="string"
                  use="required"/>
       <sequence>
         <element name="Student"
                  minOccurs="1"
                  maxOccurs="5"
                  use="required">
          <complexType>
          <attribute name="Name"
                     type="string"
                     use="required"/>
          <sequence>
           <element name="Height"
                    type="decimal"
                    use="required"/>
           <element name="Weight"
                    type="decimal"
                    use="required"/>
          </sequence>
          </complexType>
         </element><!-- Student -->
        </sequence>
       </complexType>
       </element><!-- Group -->
     </sequence>
```

```
      </complexType>
     </element><!-- Grade -->
    </sequence>
   </complexType>
  </element><!-- Class -->
</schema>
```

Appendix III

Partial listing of custom class XML.

```
<?xml version="1.0" ?>
<class:Class
  xmlns:class="http://class.org/Class">
<!-- Units Metadata -->
<units>
  <ageunit name="year"/>
  <htunit name="inch"/>
  <wtunit name="pound"/>
</units>
<!-- Class Data -->
 <Grade Age="11">
  <Group Sex="F">
   <Student Name="Joyce">
     <Height> 51.3 </Height>
     <Weight> 50.5 </Weight>
   </Student>
  </Group>
  <Group Sex="M">
   <Student Name="Thomas">
     <Height> 57.5 </Height>
     <Weight> 85 </Weight>
   </Student>
  </Group>
 </Grade>
 <Grade Age="12">

...
```

Appendix IV

XMLMap to read custom class XML.

```
<?xml version="1.0" ?>
 <SXLEMap version="1.1">
  <TABLE name="Class">
   <TABLE-PATH syntax="xpath">
     /class:Class/Grade/Group/Student
     </TABLE-PATH>
    <COLUMN name="Age" retain="yes">
     <PATH>/class:Class/Grade/@Age
     </PATH>
     <TYPE>character</TYPE>
     <DATATYPE>INTEGER</DATATYPE>
     <LENGTH>3</LENGTH>
    </COLUMN>
    <COLUMN name="Sex" retain="yes">
     <PATH>/class:Class/Grade/Group/@Sex
     </PATH>
     <TYPE>character</TYPE>
     <DATATYPE>STRING</DATATYPE>
    </COLUMN>
    <COLUMN name="Name">
     <PATH>
     /class:Class/Grade/Group/Student/@Name
     </PATH>
     <TYPE>character</TYPE>
     <DATATYPE>STRING</DATATYPE>
     <LENGTH>8</LENGTH>
    </COLUMN>
    <COLUMN name="Height">
      <PATH>
      /class:Class/Grade/Group/Student/Height
      </PATH>
      <TYPE>numeric</TYPE>
      <DATATYPE>FLOAT</DATATYPE>
    </COLUMN>
    <COLUMN name="Weight">
     <PATH>
     /class:Class/Grade/Group/Student/Weight
     </PATH>
     <TYPE>numeric</TYPE>
     <DATATYPE>FLOAT</DATATYPE>
    </COLUMN>
  </TABLE>
</SXLEMap>
```

Appendix V

Write and use a slightly modified tagset.

```
ods path sasuser.templat (update)
   sashelp.tmplmst (read);
proc template;
    define tagset tagsets.new /store=sasuser.templat;
    parent=tagsets.default;
    define event code_stylesheet_link;
      putq "      <LINK REL=""STYLESHEET"" HREF=" url "/>"
          NL / if exist( url );
   end;
end;
run;
ods xml type=new
    path='output' (url=none)
    file='class.xml'
    code='class.xsl'
    stylesheet='class.css';
proc print data=sashelp.class noobs;
   where age=14 and sex='F';
   title 'Generating HTML from XML, XSL and CSS';
run;
ods xml close;
x "cd output";
x "java org.apache.xalan.xslt.Process  \
    -XSL class.xsl -IN class.xml -HTML \
    -OUT class.html";
x "cd ..";
```