Paper 1282-2014

SAS® XML Programming Techniques

Chris Schacherer, Clinical Data Management Systems, LLC

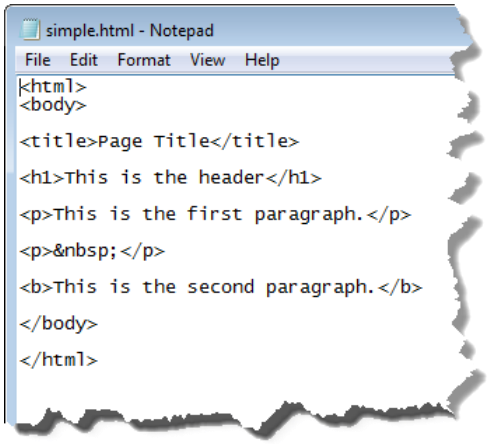
ABSTRACT

Due to XML's growing role in data interchange, it is increasingly important for SAS® programmers to become proficient with SAS technologies and techniques for creating and consuming XML. The current work expands on a SAS® Global Forum 2013 presentation that dealt with these topics—providing additional examples of using XML maps to read and write XML files and using the Output Delivery System (ODS) to create custom tagsets for generating XML.

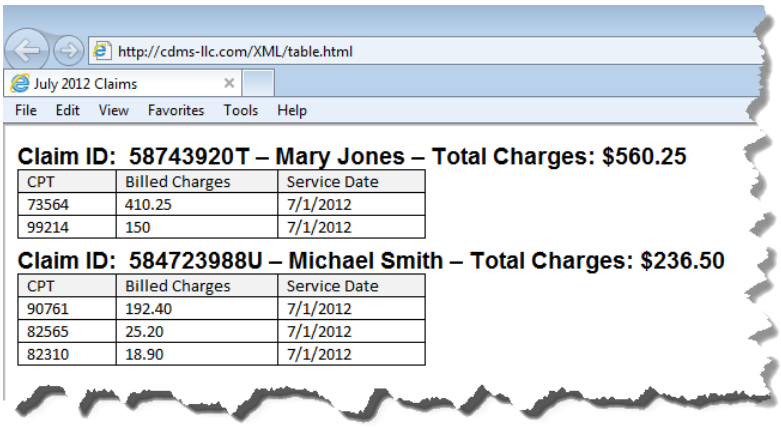
INTRODUCTION

Many tutorials on the use of SAS to produce and read XML files assume at least an introductory level of XML knowledge on the part of the SAS programmer, but many SAS programmers have no previous experience with mark-up languages—as we spend most of our time dealing with data coming to us from flat files, proprietary database systems, and the like. Many SAS analysts (the author included) have heard a mild buzz about XML for several years, but have had little if any reason in their day-to-day work to pay any attention to this "thing that is like HTML, but is something different". With the increasing presence of web services as a data source and the need to transfer data in XML format, it is high time (or even past the time) for all SAS programmers to gain at least a cursory knowledge of XML and the SAS technologies that touch it.

To begin this discussion, it is necessary to understand what Extensible Markup Language (XML) *is* and why it has gained popularity as a data transmission and integration technology. First, at a conceptual level, markup languages encompass data (and the attributes of those data) within tags that provide additional information about how those data are to be organized and understood, processed, and/or presented. For example, in the following Hypertext Markup Language (HTML) file "simple.html", the title of the web page is specified by entering the text "Page Title" inside the <title> tag, the header "This is the header" is specified using the <h1> tag and further down on the page the <b> tag is used to present the bolded text "This is the second paragraph". These tags are interpreted by your web-browser such that data (text) between these tags are rendered to the screen in specific ways according to the HTML standard. In other words, the "markup" of these data with these particular tags assures that they will be reliably rendered in this fashion across all HTML-compatible web browsers.



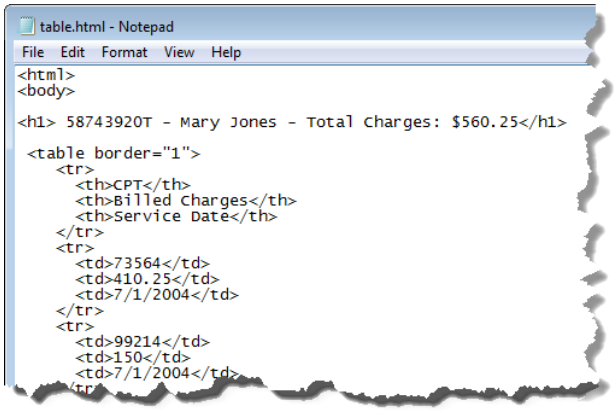
In an example more closely resembling a real-world application of using HTML to markup and present a dataset to an end-user, consider the following list of medical claims in which the claim ID, member name, and total charges serve as a header record followed by an HTML table of the line-item details associated with the header record. This presentation of the data is relatively user-friendly for someone that wants to visually inspect the line-item detail associated with a series of healthcare claims.

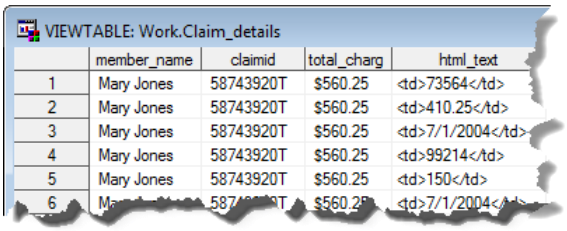


However, it is considerably less user-friendly to the data analyst who wants to analyze the claims data or generate reports and graphs other than the ones rendered in HTML. To illustrate this point, consider how one might go about programmatically extracting data about medical claims from the previous HTML file using SAS. Even in the following simplified version of the HTML file, the parsing of the "data" contained in the HTML file is fairly complex.

```
DATA claim_details;
RETAIN member_name claimid total_charges;
  INFILE clmsin truncover;
  INPUT @1 html_text $32767.;
  IF html_text = '' THEN DELETE;
  IF SUBSTR(html_text,1,4) = '<h1>' THEN DO;
   member_name = SCAN(html_text,2,'-');
       claimid = SUBSTR(SCAN(html_text,1,'-'),5);
 total_charges = SUBSTR(SCAN(html_text,3,'-'),16,LENGTH(SCAN(html_text,3,'-'))-20);
  END;
  IF SUBSTR(html_text,1,4) NE '<td>' THEN DELETE;
RUN;
```

All of the functions executed in the preceding code are needed just to get the information from the header record of each claim in the HTML file onto the corresponding record in the "claim_details" dataset. Beyond this, additional transformations would still be necessary in order to get the data from the individual HTML table columns (cpt, billed charges, service date) into discrete variables on a single row of data representing each claim line-item.





Programmatically extracting data from a file like this is fraught with peril because the individual data elements are not explicitly defined nor is the relationship between the data elements. For example, the only way that we know the table elements represent the CPT code, billed charge, and service date is through our visual inspection of the HTML code. The SAS code being written assumes that the order of the variables will be consistent across all tables in the file. Of course, you could read-in and process the column names (<th> tags) to confirm this assumption before reading each table, but that would add additional complexity to your SAS code. The bottom line here is that HTML provides a great way to present data to humans, but represents a less-than-optimal way to exchange data between computers. This is precisely where XML fits in. Whereas HTML is more focused on the presentation of the data

(e.g., should a data element be shown in a bold or italicized font face, organized in a table, etc.), XML focuses on reliably communicating the data in an unequivocal, machine-readable format.

Looking at the following XML depiction of the data from the previous example, you can see that the tags clearly identify both the data elements and their relationships to one another. Specifically, it is clear from the hierarchical organization of the data that the two "detail" elements are both associated with (or belong to) the "claim" identified by "claimid" "587439204T". Also, the names of the tags make it clear that the values "73564" and "99214" are "CPT" values associated with their respective "detail" element.

The first line of the file is the XML declaration used to identify the file as an XML file—in this case, written in compliance with version 1.0 of the XML Specification (W3C, 2008, 2006) and utilizing "WINDOWS-1252" encoding. The Root Tag "claims" describes the content of the file (i.e., Medical Claims). The second level of the hierarchy describes a "claim" and the next level describes a line-item "detail" record related to that claim. Note here that unlike the HTML tags, the hierarchical structure of the tags makes the relationship of the detail line items to the claimid unequivocal.

```
<?xml version="1.0" encoding="WINDOWS-1252"?>          ←XML Declaration
<!-- XML File for Submission of Claims Data -->        ←Comments
-<medical claims>                                      ←Root Element (opening tag)
    -<claim>
     <claimid>587439204T</claimid>
        <detail>
            <cpt>73564</cpt>
            <acct_no>VGH3344562</acct_no>
            <member>Jones, Mary</member>
            <billed_charges>410.25</billed_charges>    • Child elements describing
        </detail>                                        "claims"
        <detail>
            <cpt>99214</cpt>
            <acct_no>VGH3344562</acct_no>
            <member>Jones, Mary</member>
            <billed_charges>150.00</billed_charges>
        </detail>
    </claim>
</medical claims>                                      ←Root Element (closing tag)
```

This example also shows us what it means when XML is described as both "self-describing" and "extensible". The file is "self-describing" in that by nesting tags (and/or by assigning attributes to the tagged values) you can describe the relationships of data elements to one another. This is an important advantage over flat files when exchanging data from relational data structures[1]. The data elements themselves are self-describing in that you can create descriptive names for the tags (e.g., "claimid", "account_no", etc.). This latter point also exemplifies the main way in which XML is "extensible". Consider the tags in HTML (e.g., /TABLE, /bold /p, /heading); these tags are defined as part of the HTML specification and are immutable if one wants their HTML to display correctly. The tagset names in XML, on the other hand, are completely under the control of the individual creating the markup file; you are, in essence, creating your own specific markup language when you define an XML file's architecture. In this "claims" file, for example, a claims markup language for transmission of medical claims data is created—wherein tags for "claimid", "account_no", and "member" explicitly identify the type of content being communicated. Of course, whether this markup language is adopted by others in your company or broader industry and becomes a "standard" is another

[1] Although outside the scope of the current paper, through utilization of Data Type Definitions (DTDs) and XML Schemas (written in the XML Schema Definition, or XSD, language) the individual data elements can be further defined in terms of their data type and restricted to specific value ranges, acceptable discrete values, and conditional logic criteria.

story. Nonetheless, using XML, you can create a custom file architecture that clearly and unequivocally identifies the data values being communicated as well as their relationship to one another[2].

This facility to clearly communicate data elements from even very complicated data structures is a key factor in XML's popularity. If XML were a proprietary commercial "product", the self-describing and extensible nature of the language would be enough to make it a big seller, but XML also has the added benefit of being non-proprietary and platform independent! As a result, XML has gained wide-spread acceptance as a go-to technology for exchanging data—both in the form of producing XML files for batch submissions and in the exchange of data via web services. SAS has developed a number of technologies to allow you to take advantage of the power and flexibility of XML, but before those are discussed you should know the basic rules governing the structure of an XML file.

The first line of any XML file is the XML declaration (which is used to identify the file as an XML file). In the preceding example, you can see that the file is written in compliance with version 1.0 of the XML Specification (W3C, 2008, 2006) and that it utilizes "WINDOWS-1252" encoding. The Root Tag "claims" describes the content of the file (i.e., Medical Claims). The second level of the hierarchy describes a "claim" and the next level describes a line-item "detail" record related to that claim. Note again that unlike the HTML tags, the hierarchical structure of the XML tags defines the relationships between the data elements.

In addition to the requirement of a declaration statement and a single root tag, there are a few other characteristics that an XML file must have in order to be "well-formed"—and note, that only well-formed files will be capable of being read by XML-compliant parsers. First, according to Castro & Goldberg (2009), the root tag must contain all other tags in the file; that is, no tags are allowed before the root tag is opened or after the root tag is closed. Each tag (e.g., "<claim>") must have a closing tag ("</claim>")—although, empty tags can be closed by the same tag that opens them (e.g., "<claim />"). And tags cannot cross each other's boundaries. For example, the file containing the following XML elements is not well-formed because the tag that closes "demographics" crosses the boundary of the "claims" element in which the demographics element is opened.

```
<claims>
    <demographics>
    <firstname>Mary</firstname>
    <lastname>Jones</lastname>
</claims>
    <gender>F</gender>
    </demographics>
```

Also, XML tags are case sensitive, so "<claims>" identifies a completely different tag than "<Claims>" or "<CLAIMS>". XML elements can contain both values (e.g., claimid "587439204T") and attributes which further describe the element. In the following example, "unit" is an attribute that describes the unit of measurement associated with the value (110) of the "weight" element:

```
<weight unit="KG">110</weight>
```

Finally, like HTML, white space is ignored by XML parsers, so you can add white space to make XML files easier to read.

With this basic understanding of XML, you are ready to start exploring the various technologies that SAS has developed to help you interact with these powerful data structures. The first of these technologies that we will explore is the XML LIBNAME Engine.

XML LIBNAME ENGINE OVERVIEW

The XML LIBNAME engine, like the other SAS LIBNAME engines, provides a means of accessing data in formats other than a SAS dataset. Just as you can execute a LIBNAME statement to operate against data in an Oracle®, SQL Server®, or DB2® database, so too can you define a SAS library to operate against an XML file. Note that normally, when using a LIBNAME statement in this way, you are reading/writing to proprietary software—e.g., Oracle, etc.—and the job of the SAS data engine is to translate between SAS and the proprietary data format. Similarly,

[2] However, as Cox (2012) points out, common misunderstandings about what it means to be "extensible", "self-describing", and a "standard" are responsible for much of the confusion about XML. His description of what XML "is" and what it "is not" is recommended reading if you are just getting started with XML in SAS.

even though an XML file is essentially little more than a text file, the XML engine acts on behalf of the analytic engine to determine the dataset name(s), structure(s), and individual data elements contained in the specified file. By default, the XML LIBNAME engine identifies the root element as the container (or source library) of the data and the second-level elements (or, nodes) as the "datasets".

In the following example, a LIBNAME statement is used to define the SAS library "clmfile" as a library that utilizes the SAS XML engine to access data stored in the "claim_detail.xml" file.

```
LIBNAME clmfile XML '\\datasrvr\inbound\claim_detail.xml';
```

As depicted, below, this file (claim_detail.xml) contains data about individual line-item records associated with a series of medical claims. Once the file is "declared" as an XML file ❶ and the root tag is specified ❷, the "detail" elements are used as containers for line-item detail records ❸.

When the XML engine is invoked to access this file as the SAS library "clmfile", SAS recognizes that "claims" is merely a root tag acting as a container for the data of interest—the "detail" elements. In other words, what the dataset is really "about" are the line-item details associated with the medical claims. Although marked up as individual tag elements in the XML file, you can think of each "detail" element as representing a "record" in a claims "dataset". This is precisely how the XML engine represents these data to SAS DATA and PROC steps.

```
<?xml version="1.0" encoding="WINDOWS-1252"?> ❶
<!-- XML File for Submission of Claims Data -->
❷ <claims xmlns="http://www.cdms-llc.com/ns/sasclaims/1.0">
  ❸ <detail>
      <claimid>584723988U</claimid>
      <line>1</line>
      <acct_no>XWY3928957</acct_no>
      <member>Smith, Michael</member>
      <billed_charges>192.40</billed_charges>
      <total_charges>236.50</total_charges>
    </detail>
  ❸ <detail>
      <claimid>584723988U</claimid>
      <line>2</line>
      <acct_no>XWY3928957</acct_no>
      <member>Smith, Michael</member>
      <billed_charges> 25.20</billed_charges>
      <total_charges>236.50</total_charges>
    </detail>
  ❸ <detail>
      <claimid>584723988U</claimid>
      <line>3</line>
      <acct_no>XWY3928957</acct_no>
      <member>Smith, Michael</member>
      <billed_charges> 18.90</billed_charges>
      <total_charges>236.50</total_charges>
    </detail>
  ❸ <detail>
      <claimid>587439204T</claimid>
      <line>1</line>

    ...
```

For example, in the following DATA step, a local copy of the claims data is created as a SAS dataset.

```
LIBNAME local 'c:\_data\local\';
LIBNAME clmfile XML '\\datasrvr\inbound\claim_detail.xml';

DATA local.claim_detail;
 SET clmfile.detail;
RUN;
```

Note again that the "dataset" being accessed in the clmfile library has the same name as the first level "child" tag of the XML file and that (as depicted below) each of the records in the dataset "local.claim_detail" corresponds to one of the "detail" nodes in the "claim_detail.xml" file. One interesting characteristic of the SAS file is that the order of the variables is reversed compared to the order in which their corresponding elements appear within the "detail" elements. This suggests that SAS is reading all of the data for each "detail" element before the data for that element are written to the SAS file record. This is reasonable in that SAS does not know that the end of a "record" has been reached until the closing tag is encountered.

VIEWTABLE: Local.Claim_detail

	TOTAL_CHARGES	BILLED_CHARGES	MEMBER	ACCT_NO	LINE	CLAIMID
1	236.5	192.4	Smith, Michael	XWY3928957	1	584723988U
2	236.5	25.2	Smith, Michael	XWY3928957	2	584723988U
3	236.5	18.9	Smith, Michael	XWY3928957	3	584723988U
4	560.25	410.25	Jones, Mary	VGH3344562	1	587439204T
5	560.25	150	Jones, Mary	VGH3344562	2	587439204T
6	636.52	383.12	Roberts, Stuart	JKL6857483	1	866978852B
7	636.52	192	Roberts, Stuart	JKL6857483	2	866978852B
8	636.52	18.9	Roberts, Stuart	JKL6857483	3	866978852B
9	636.52	42.5	Roberts, Stuart	JKL6857483	4	866978852B

Just as you can read an XML file using a DATA step against an XML library, you can also write data from SAS to an XML file. In the following example, the dataset "local.claim_detail" from the previous example is written out to the xml file "claim detail out.xml".

```
LIBNAME clmout XML '\\datasrvr\outbound\claim detail out.xml';

DATA clmout.detail;
 SET local.claim_detail;
RUN;
```

The resulting file, below, does not look exactly like the original source dataset (note, for example that all of the tag names are in upper case and the default root tag is declared as "TABLE"), but the contents of the SAS dataset are represented in the same simple XML structure as the original XML source file.

```
<?xml version="1.0" encoding="WINDOWS-1252"?>
- <TABLE>
  - <DETAIL>
      <TOTAL_CHARGES> 236.5 </TOTAL_CHARGES>
      <BILLED_CHARGES> 192.4 </BILLED_CHARGES>
      <MEMBER> Smith, Michael </MEMBER>
      <ACCT_NO> XWY3928957 </ACCT_NO>
      <LINE> 1 </LINE>
      <CLAIMID> 584723988U </CLAIMID>
    </DETAIL>
  - <DETAIL>
      <TOTAL_CHARGES> 236.5 </TOTAL_CHARGES>
      <BILLED_CHARGES> 25.2 </BILLED_CHARGES>
      <MEMBER> Smith, Michael </MEMBER>
      <ACCT_NO> XWY3928957 </ACCT_NO>
      <LINE> 2 </LINE>
      <CLAIMID> 584723988U </CLAIMID>
    </DETAIL>
  - <DETAIL>
      <TOTAL_CHARGES> 236.5 </TOTAL_CHARGES>
      <BILLED_CHARGES> 18.9 </BILLED_CHARGES>
      <MEMBER> Smith, Michael </MEMBER>
      <ACCT_NO> XWY3928957 </ACCT_NO>
      <LINE> 3 </LINE>
      <CLAIMID> 584723988U </CLAIMID>
    </DETAIL>
  - <DETAIL>
      <TOTAL_CHARGES> 560.25 </TOTAL_CHARGES>
      <BILLED_CHARGES> 410.25 </BILLED_CHARGES>
      <MEMBER> Jones, Mary </MEMBER>
      <ACCT_NO> VGH3344562 </ACCT_NO>
      <LINE> 1 </LINE>
      <CLAIMID> 587439204T </CLAIMID>
```

Hopefully from these first simple examples, you are starting to get a feel for how the XML LIBNAME engine operates. When reading XML files, SAS reads the tagset and based on the tags it finds, creates a dataset, names it, and creates, names, and values the variables associated with each second-level element in the file.

Even if the data in the other detail elements are written in different orders (as in the two datasets below), the dataset

will retain the order defined when the first detail element is read from the file. This is because the tags (not their order within the detail element) are used to determine the content of a given SAS dataset variable.[3] The following two XML files result in identical SAS datasets.

```
<?xml version="1.0" encoding="WINDOWS-1252"?>
<!-- XML File for Submission of Claims Data -->
- <claims xmlns="http://www.cdms-llc.com/ns/sasclaims/1.0">
  - <detail>
      <claimid>584723988U</claimid>
      <line>1</line>
      <acct_no>XWY3928957</acct_no>
      <member>Smith, Michael</member>
      <billed_charges>192.40</billed_charges>
      <total_charges>236.50</total_charges>
    </detail>
  - <detail>
      <line>2</line>
      <claimid>584723988U</claimid>
      <billed_charges> 25.20</billed_charges>
      <acct_no>XWY3928957</acct_no>
      <member>Smith, Michael</member>
      <total_charges>236.50</total_charges>
    </detail>
  - <detail>
      <billed_charges> 18.90</billed_charges>
      <total_charges>236.50</total_charges>
      <claimid>584723988U</claimid>
      <line>3</line>
      <acct_no>XWY3928957</acct_no>
      <member>Smith, Michael</member>
    </detail>
  - <detail>
      <claimid>587439204T</claimid>
      <line>1</line>
```

```
<?xml version="1.0" encoding="WINDOWS-1252"?>
<!-- XML File for Submission of Claims Data -->
- <claims xmlns="http://www.cdms-llc.com/ns/sasclaims/1.0">
  - <detail>
      <claimid>584723988U</claimid>
      <line>1</line>
      <acct_no>XWY3928957</acct_no>
      <member>Smith, Michael</member>
      <billed_charges>192.40</billed_charges>
      <total_charges>236.50</total_charges>
    </detail>
  - <detail>
      <claimid>584723988U</claimid>
      <line>2</line>
      <acct_no>XWY3928957</acct_no>
      <member>Smith, Michael</member>
      <billed_charges> 25.20</billed_charges>
      <total_charges>236.50</total_charges>
    </detail>
  - <detail>
      <claimid>584723988U</claimid>
      <line>3</line>
      <acct_no>XWY3928957</acct_no>
      <member>Smith, Michael</member>
      <billed_charges> 18.90</billed_charges>
      <total_charges>236.50</total_charges>
    </detail>
  - <detail>
      <claimid>587439204T</claimid>
      <line>1</line>
```

The flexibility of XML, however, can lead to some unexpected results if you are not familiar with your data and the specification of the particular XML file with which you are working. When you are working with an Excel® file or an Oracle table, you are used to the fact that for each record there can be one and only one value for a given variable. In XML this basic premise of our data management reality can seem to stretch a bit. In the following example, a medical claims file has more than one procedure associated with each "claimid".

```
<?xml version="1.0" encoding="WINDOWS-1252"?>
<!-- XML File for Submission of Claims Data -->
- <claims xmlns="http://www.cdms-llc.com/ns/sasclaims/1.0">
  - <claim>
      <claimid>584723988U</claimid>
      <cpt>73564</cpt>
      <cpt>99214</cpt>
      <acct_no>XWY3928957</acct_no>
      <member>Smith, Michael</member>
      <total_charges>236.50</total_charges>
    </claim>
  - <claim>
      <claimid>584723988U</claimid>
      <cpt>90761</cpt>
      <cpt>82565</cpt>
      <cpt>82310</cpt>
      <acct_no>XWY3928957</acct_no>
      <member>Jones, Mary</member>
      <total_charges>827.39</total_charges>
    </claim>
</claims>
```

When these data are read in with the following DATA step, the results are not entirely surprising. As each additional piece of information with the same tag ("<cpt>") is read into the SAS data record (before the closing "</claim>" tag is encountered), the XML libname engine must be capable of handling the repeated tags. In this case, an integer is appended to the name of the tag name as each additional instance of that tag is discovered within the same parent tag—resulting in the following dataset.

[3] Conversely, why an XML file would be written in this sort of random order in the first place is a bit of a mystery, but the point remains—the corresponding SAS variable values are consistently mapped based on the XML tag with the same name.

```
LIBNAME clmfile XML '\\datasrvr\inbound\claim_detail - MULTIPLE.xml';

DATA local.claim_detail_multiple;
 SET clmfile.claims;
RUN;
```

VIEWTABLE: Local.Claim_detail_multiple

	CPT2	TOTAL_CHARGES	MEMBER	ACCT_NO	CPT1	CPT0	CLAIMID
1	.	236.5	Smith, Michael	XWY3928957	99214	73564	584723988U
2	82310	827.39	Jones, Mary	XWY3928957	82565	90761	584723988U

The preceding, valid XML file exemplifies both the power of XML to maintain information about relational data structures and the challenges inherent in converting such a multidimensional architecture into a two dimensional dataset. Because an infinite number of unique markup "languages" could be developed to describe our data, we really are asking the XML libname engine to take on quite a challenging task. Can you really expect the XML engine to read in all possible XML files and produce a single two-dimensional (rows x columns) dataset? There are limits on the complexity of the XML files that the XML LIBNAME engine can handle in its default, native state. However, as you will see shortly, SAS does provide additional functionality to expand the file complexity that the XML LIBNAME engine can handle. Before discussing techniques for reading more complex XML structures, however, there are a number of XML LIBNAME options of which you should be aware.

XML LIBNAME OPTIONS

As with other SAS/ACCESS engines, XML LIBNAMEs can be configured in a number of ways using LIBNAME options. Though the following is not an exhaustive list of options available for the XML ACCESS engine, some important variants on the XML LIBNAME statement are described below.

NICKNAMES – XML & XMLV2

Unlike other SAS/Access engines—in which differences in the engine's functionality depends not on a "version" of the access engine itself but on the back-end datasource to which one is connecting—the XML Access engine does have two different "versions" referred to by their nicknames. The "XML" access engine nickname invokes functionality that was available in SAS version 9.1.3, and the "XMLV2" nickname invokes functionality that was implemented subsequent to SAS 9.1.3. Though for a great many XML-related tasks the two nicknames are interchangeable, there are a few important differences with which you should be familiar.

First, whereas the XML nickname allows XML libraries to be assigned against files that may violate the XML specification, the XMLV2 nickname is XML compliant. For example, looking at the following XML segment, you will notice that the closing tag "TOTAL_CHARGES" does not match the opening tag "total_charges"—and thus, the XML is not well-formed.

```
<CLAIM_DETAIL>
  <claim_id>587439204T</claim_id>
  <total_charges>560.2</TOTAL_CHARGES>
  <svc_date>2012-07-01</svc_date>
  <account_no>VGH3344562</account_no>
  <member>Jones, Mary</member>
  <cpt>99214</cpt>
  <billed_charges>150</billed_charges>
</CLAIM_DETAIL>
```

Despite this, the following LIBNAME assignment is executed successfully using the XML nickname.

```
LIBNAME claims XML '\\datasrvr\inbound\claims-invalid xml.xml';

172  LIBNAME claims XML '\\datasrvr\inbound\claims-invalid xml.xml';
NOTE: Libref claims was successfully assigned as follows:
      Engine:        XML
      Physical Name: \\datasrvr\inbound\claims-invalid xml.xml
```

Using the XMLV2 engine, however, the XML specification violations are enforced and the "claims" library assignment fails:

```
LIBNAME claims XMLV2 '\\datasrvr\inbound\claims-invalid xml.xml';

173  LIBNAME CLAIMS XMLV2 'C:\SGF 2014\XML\claims-invalid xml.xml';
ERROR: Unexpected or unmatched end of element tag
       occurred at or near line 5, column 45
ERROR: XML parsing error. Please verify that the XML content is well-formed.
ERROR: Error in the LIBNAME statement.
```

To add further potential confusion to this situation, despite successfully assigning the LIBNAME "claims" with the XML nickname, a DATA step attempting to read the data from the library fails with an "unmatched tag" error. Although the XML nickname allows the library to be assigned, processing the XML data still results in an error.

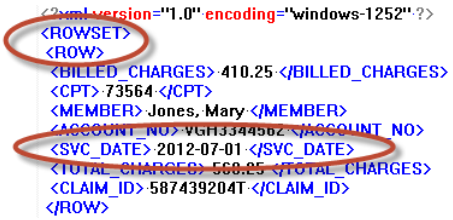
```
LIBNAME claims XML '\\datasrvr\inbound\claims-invalid xml.xml';

DATA work.claims;
 SET claims.claim_detail;
RUN;

179  DATA work.claims;
180   SET claims.claim_detail;
ERROR: Unexpected or unmatched end of element tag
ERROR: Encountered during XMLMap parsing at or near line 5, column 45.
ERROR: XML describe error: Internal processing error.
181  RUN;
```

Despite situations like this one, there are LIBNAME options associated with the XML nickname—mainly those that support the Clinical Data Interchange Standards Consortium (CDISC) Operational Data Model (ODM)—that are available only with the XML nickname and not with XMLV2. Beyond the many CDISC-related options (which are not discussed further in the current work), one LIBNAME option that may prompt you to use the XML nickname is the "XMLTYPE".

XMLTYPE

Although the XML language is intended to be a flexible means of transmitting and integrating data that is free from "ownership" by any individual organization, several organizations have leveraged this flexibility to generate their own standardized tagsets that their tools rely on to utilize XML data. To facilitate interaction with these tagging conventions, an XML LIBNAME can specify some specific XML "Types". For example, the following XML file is generated so that it will be written with Oracle's tagging scheme which names the root tag "<ROWSET>" and tags each of the subordinate records as "<ROW>"s of data.

```
LIBNAME CLAIMS XML '\\datasrvr\outbound\oracle claims.xml' XMLTYPE=ORACLE;

DATA CLAIMS.CLAIM_DETAIL;
 SET LOCAL.CLAIM_DETAIL;
RUN;
```



Similarly, the MSACCESS default tagging strategy uses "<dataroot>" as the root tag, but does not impose similar restrictions on the subordinate tags within the root tag. Note also that, by default under this XMLTYPE, dates are translated to their datetime equivalent.

```
LIBNAME CLAIMS XML '\\datasrvr\outbound\msaccess claims.xml' XMLTYPE=MSACCESS;

DATA CLAIMS.CLAIM_DETAIL;
 SET LOCAL.CLAIM_DETAIL;
RUN;
```

```
<?xml version="1.0" encoding="windows-1252" ?>
<dataroot>
  <CLAIM_DETAIL>
    <BILLED_CHARGES>410.25</BILLED_CHARGES>
    <CPT>73564</CPT>
    <MEMBER>Jones, Mary</MEMBER>
    <ACCOUNT_NO>VGH3344562</ACCOUNT_NO>
    <SVC_DATE>2012-07-01T00:00:00</SVC_DATE>
    <TOTAL_CHARGES>560.25</TOTAL_CHARGES>
    <CLAIM_ID>587439204T</CLAIM_ID>
  </CLAIM_DETAIL>
```

The first example of XML output provided in the current work used the default "GENERIC" XMLTYPE which uses the "<TABLE>" root tag. Because GENERIC is the default XMLTYPE, it is not necessary to specify the XMLTYPE option, although doing so will make explicit your intention to use the GENERIC XMLTYPE rather than some other type.

```
LIBNAME CLAIMS XML '\\datasrvr\outbound\XML Generic Claims.xml' XMLTYPE=GENERIC

DATA CLAIMS.CLAIM_DETAIL;
 SET WORK.CLAIM_DETAIL;
RUN;
```

```
<?xml version="1.0" encoding="windows-1252" ?>
<TABLE>
  <CLAIM_DETAIL>
    <claim_id>587439204T</claim_id>
    <total_charges>560.25</total_charges>
    <svc_date>2012-07-01</svc_date>
    <account_no>VGH3344562</account_no>
    <member>Jones, Mary</member>
    <cpt>73564</cpt>
    <billed_charges>410.25</billed_charges>
  </CLAIM_DETAIL>
```

XMLFILEREF

Although all of the examples so far have specified the path and filename to the XML file within the definition of the LIBNAME, the LIBNAME can also reference the XML source/destination by referring to a logical FILNAME. This is especially useful processing XML written to temporary files as might be done when interacting with web services (see, for example, Schacherer 2012).

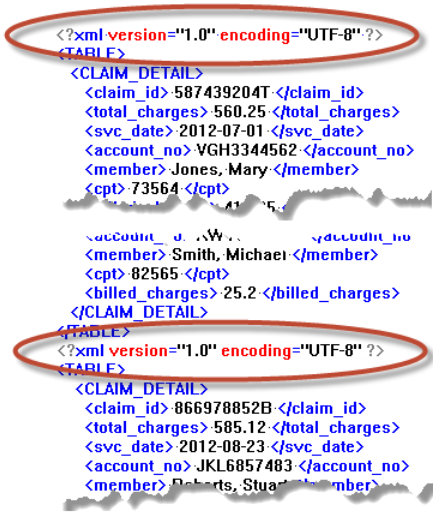
```
FILENAME INFILE TEMP;
LIBNAME CLAIMS XML XMLFILEREF=INFILE;

DATA claims.claim_detail ;
 SET work.claim_detail;
RUN;
```

You may also see LIBNAMES assigned to a logical filename without the XMLFILEREF option. Although this LIBNAME assignment works perfectly well, it can be confusing because the relationship between the LIBNAME and the FILENAME is not explicitly stated. In the following example, the "OUTFILE" LIBNAME still successfully writes to the "INFILE" FILENAME, but because there is no explicit assignment to a filename, the LIBNAME statement can be confusing—especially if the FILENAME statement does not immediately precede the LIBNAME statement.

```
FILENAME OUTFILE TEMP;
LIBNAME OUTFILE XML;

DATA outfile.claim_detail ;
 SET work.claim_detail;
RUN;
```

XMLCONCATENATE

Although it has been stressed so far that XML must be well-formed in order to be reliably read through the XML LIBNAME ENGINE, the XMLCONCATENATE option specifically modifies the LIBNAME in order to read one type of file that is purposefully not well-formed. In the following example, multiple claims files are concatenated together in a single batch processing file of all submitted claims files from July and August 2012. Note that there are muliple XML declarations and multiple root tags (<TABLE>/</TABLE>) within this file.



It is not well-formed XML, but by sepecifying the XMLCONCATENATE=YES option, the DATA step successfully reads the file—parsing the records from both XML files into a single dataset. One should also note that according to SAS (2012), "This option is included for convenience, not to encourage invalid XML markup", p. 35.

```
LIBNAME CLAIMS XML '\\datasrvr\inbound\Claims Double.xml' XMLCONCATENATE=YES;

DATA work.claim_detail ;
 SET claims.claim_detail;
RUN;
```

TABLE: Work.Claim_detail

BILLED_CHARGES	CPT	MEMBER	ACCOUNT_NO	SVC_DATE	TOTAL_CHARGES	CLAIM_ID
410.25	73564	Jones, Mary	VGH3344562	2012-07-01	560.25	587439204T
150	99214	Jones, Mary	VGH3344562	2012-07-01	560.25	587439204T
192.4	90761	Smith, Michael	XWY3928957	2012-07-01	217.6	584723988U
25.2	82565	Smith, Michael	XWY3928957	2012-07-01	217.6	584723988U
383.12	73564	Roberts, Stuar	JKL6857483	2012-08-23	585.12	866978852B
202	73564	Roberts, Stuar	JKL6857483	2012-08-23	585.12	866978852B
25.2	82565	Smith, Michael	XWY3928957	2012-08-05	44.1	829925635A
18.9	82310	Smith, Michael	XWY3928957	2012-08-05	44.1	829925635A
403.52	73564	Roberts, Stuar	JKL6857483	2012-08-12	1003.8	1225639533
200.03	73564	Roberts, Stuar	JKL6857483	2012-08-12	1003.8	1225639533

So far, in discussing the XML LIBNAME, we have discussed the XML nicknames and how different LIBNAME options can alter the mechanics of how the XML LIBNAME accesses XML files and enforces the XML specification (or not). But what we have not discussed is how the XML LIBNAME "understands" the many different ways in which XML files can be structured.

XML RULES FOR DEFAULT LIBNAME ACCESS

Having considered some of the different options for using the XML LIBNAME engine to interact with XML files, there are also some basic rules governing the default capability of the XML LIBNAME engine to read data from (or write data to) an XML file.

To better understand this capability, you should note that the GENERIC XMLTYPE is capable of reliably reading only those XML files with the following characteristics (SAS, 2010): (1) the top-level, or "root" node is the document container—that is, the root node contains all other nodes. This is also one of the requirements for well-formed XML. (2) the repeated elements that correspond to the definition of the "records" in the preceding examples must begin at the second-level in the XML file's hierarchy. (3) The contents of the second-level, repeating element representing the "records" must represent a "rectangular" (rows x columns) organization of the data they contain. That is, within the repeating tag, all additional elements must be at the same level of the hierarchy.

In the following example, the values of the "cpt" tags are all at the same level in the hierarchy, but are not the second level of the element hierarchy.

```
<?xml version="1.0" encoding="WINDOWS-1252"?>
<!-- XML File for Submission of Claims Data -->
- <claims xmlns="http://www.cdms-llc.com/ns/sasclaims/1.0">
  - <claim>
      <claimid>584723988U</claimid>
    - <cpts>
        <cpt>73564</cpt>
        <cpt>99214</cpt>
      </cpts>
      <acct_no>XWY3928957</acct_no>
      <member>Smith, Michael</member>
      <total_charges>236.50</total_charges>
    </claim>
  - <claim>
      <claimid>584723988U</claimid>
    - <cpts>
        <cpt>90761</cpt>
        <cpt>82565</cpt>
        <cpt>82310</cpt>
      </cpts>
      <acct_no>XWY3928957</acct_no>
      <member>Jones, Mary</member>
      <total_charges>827.39</total_charges>
    </claim>
</claims>
```

As a result, although the following DATA step does not generate errors when executed, the resulting SAS dataset contains missing values for the "cpts" and "cpt<x>" variables.

```
LIBNAME clmfile XML '\\datasrvr\inbound\claim_detail.xml' XMLTYPE=GENERIC;

DATA local.claim_detail_three_levels;
 SET clmfile.claim;
RUN;
```

VIEWTABLE: Local.Claim_detail_three_levels

	TOTAL_CHARGES	MEMBER	ACCT_NO	CPT2	CPT1	CPTS	CPT0	CLAIMID
1	236.5	Smith, Michael	XWY3928957	.	.		.	584723988U
2	827.39	Jones, Mary	XWY3928957	.	.		.	584723988U

As the hierarchy of the XML file becomes more complex, as in the following example, the XML LIBNAME engine does not even attempt to translate the file into a SAS dataset, and generates an error to the log.

```
<?xml version="1.0" encoding="WINDOWS-1252"?>
<!-- XML File for Submission of Claims Data -->
- <claims xmlns="http://www.cdms-llc.com/ns/sasclaims/1.0">
  - <claim>
      <claimid>584723988U</claimid>
    - <detail>
        <total_charges>236.50</total_charges>
        <line>1</line>
        <acct_no>XWY3928957</acct_no>
        <member>Smith, Michael</member>
        <billed_charges>192.40</billed_charges>
      </detail>
    - <detail>
        <total_charges>236.50</total_charges>
        <line>2</line>
        <acct_no>XWY3928957</acct_no>
        <member>Smith, Michael</member>
        <billed_charges> 25.20</billed_charges>
      </detail>
    - <detail>
        <total_charges>236.50</total_charges>
        <line>3</line>
        <acct_no>XWY3928957</acct_no>
        <member>Smith, Michael</member>
        <billed_charges> 18.90</billed_charges>
      </detail>
    </claim>
```

```
LIBNAME clmfile XML '\\datasrvr\inbound\claim_detail.xml' XMLTYPE=GENERIC;

DATA local.claim_detail_complex;
 SET clmfile.claim;
RUN;


ERROR: XML describe error: XML data is not in a format supported natively by the XML
libname engine. Files of this type usually require an XMLMap to be input properly:
\\datasrvr\inbound\claim_detail.xml.
```

As these examples show, once the complexity of the file exceeds the default tolerances around which the XML LIBNAME engine is designed, accessing data from XML files becomes more challenging. Expecting the XML LIBNAME engine to be capable of determining "which" of all the possible two-dimensional datasets one might want to extract from these more complex XML structures is just not realistic. Instead, SAS utilizes XML MAP files to assist you in in defining the datasets to be extracted from these complex data sources.

XML MAPS AND XML MAPPER

As described elsewhere (Martell, 2008; SAS, 2010a; 2011a ; Shoemaker, 2005), an XML Map is a file that provides additional instructions to the LIBNAME engine in order to translate a particular XML schema into one or more SAS datasets[4]. In the following example, healthcare claims data from the XML file in the preceding example are successfully written to the SAS datasets "local.mapped_line_items" and "local.mapped_claims". The DATA steps used to perform these operations succeed in this case because the library definition for "clmfile" includes the XMLMAP option and a reference to a map file "claim translator.map" that is used by the LIBNAME engine to parse the XML into one or more predefined datasets.

```
LIBNAME local 'c:\_data\local\';

LIBNAME clmfile XML '\\datasrvr\claim_detail.xml' XMLMAP='c:\claim translator.map';
DATA local.mapped_line_items;
 SET clmfile.detail;
RUN;
```

[4] Shoemaker (2005) provides a particularly good description of both (a) the XML Map's role in acting as the translation table between the XML source file and the LIBNAME engine and (b) the mechanics of building an XML Map file.

```
DATA local.mapped_claims;
 SET clmfile.claim;
RUN;
```

VIEWTABLE: Local.Mapped_line_items

	line	billed_charges	claimid
1	1	192.40	584723988U
2	2	25.20	584723988U
3	3	18.90	584723988U
4	1	410.25	587439204T
5	2	150.00	587439204T
6	1	383.12	866978852B
7	2	192.00	866978852B
8	3	18.90	866978852B
9	4	42.50	866978852B

VIEWTABLE: Local.Mapped_claims

	claimid	total_charges	acct_no	member
1	584723988U	236.5	XWY3928957	Smith, Michael
2	587439204T	560.25	VGH3344562	Jones, Mary
3	866978852B	636.52	JKL6857483	Roberts, Stuart

So, what is in this "claim translator.map" file that enables the LIBNAME engine to translate the XML file "claim_detail.xml" into the datasets "local.mapped_claims" and "local.mapped_line_items"? It may surprise you to discover that the .map file, itself, is an XML file. This is not entirely unexpected, as one of the frequent uses of XML files is to control system configurations. In this case, the XML data in the .map file is used by the LIBNAME engine to define the XPATH locations of different SAS dataset components within the source XML file. In looking at the "claim translator.map" file, below, we see that, like any XML file, it begins with the XML file declaration ❶. The root tag for all SAS XML maps is the tag "SXLEMAP"❷. This root element contains all of the instructional parameters that tell the LIBNAME engine how to define collections of data "rows" that are to be arranged in "tables" (or, SAS datasets). Specifically, in this example, the tag "TABLE" has an attribute called "name" that is valued with "claim" ❸. As part of this table definition, a PATH tag specifies the location (in the source XML file) of the element that indicates the starting position of the elements that are to be included in the table definition. In this case the XPATH location "/claims/claim" indicates to the LIBNAME engine that the "<claim>" tag in the source file indicates the starting position of the data that will be included in the "claim" dataset ❹. Similarly, the XPATH locations of each column are specified within each "COLUMN" element ❺, and the name of each column is assigned via the "name" attribute ❻. Data types and lengths of the variables can also be assigned by elements within each COLUMN tag ❼.

```
<?xml version="1.0" encoding="WINDOWS-1252"?> ❶
❷ - <SXLEMAP version="1.2">
    - <TABLE name="claim"> ❸
        <TABLE-DESCRIPTION>claim</TABLE-DESCRIPTION>
        <TABLE-PATH syntax="XPath">/claims/claim</TABLE-PATH> ❹
      - <COLUMN name="claimid"> ❻
          <PATH syntax="XPath">/claims/claim/claimid</PATH> ❺
          <TYPE>character</TYPE>
          <DATATYPE>string</DATATYPE> ❼
          <LENGTH>32</LENGTH>
        </COLUMN>
      - <COLUMN name="total_charges">
          <PATH syntax="XPath">/claims/claim/detail/total_charges</PATH>
          <TYPE>numeric</TYPE>
          <DATATYPE>double</DATATYPE>
        </COLUMN>
      - <COLUMN name="acct_no">
          <PATH syntax="XPath">/claims/claim/detail/acct_no</PATH>
          <TYPE>character</TYPE>
          <DATATYPE>string</DATATYPE>
          <LENGTH>10</LENGTH>
        </COLUMN>
      - <COLUMN name="member">
          <PATH syntax="XPath">/claims/claim/detail/member</PATH>
          <TYPE>character</TYPE>
          <DATATYPE>string</DATATYPE>
          <LENGTH>15</LENGTH>
        </COLUMN>
    </TABLE>
...
```

Similarly, this file continues (below) with the specification of the "detail" table and ends with the closing tag of the "SXLEMAP" element.

...

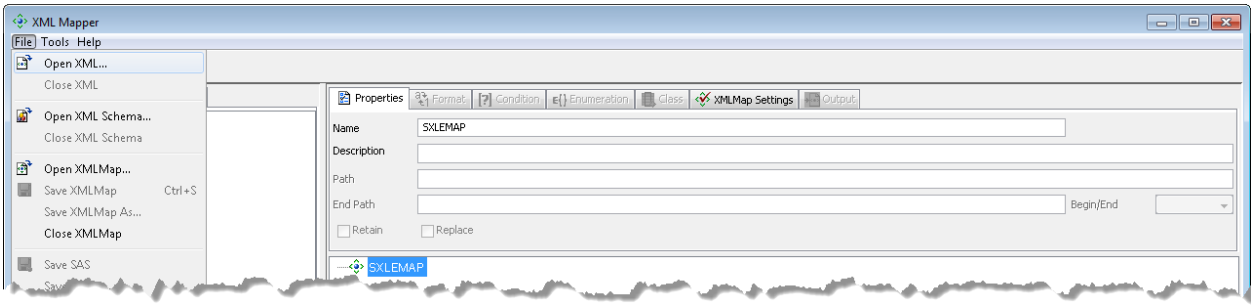
```
- <TABLE name="detail">
    <TABLE-DESCRIPTION>detail</TABLE-DESCRIPTION>
    <TABLE-PATH syntax="XPath">/claims/claim/detail</TABLE-PATH>
  - <COLUMN name="claimid" retain="YES">
      <PATH syntax="XPath">/claims/claim/claimid</PATH>
      <TYPE>character</TYPE>
      <DATATYPE>string</DATATYPE>
      <LENGTH>10</LENGTH>
    </COLUMN>
  - <COLUMN name="line">
      <PATH syntax="XPath">/claims/claim/detail/line</PATH>
      <TYPE>character</TYPE>
      <DATATYPE>string</DATATYPE>
      <LENGTH>32</LENGTH>
    </COLUMN>
  - <COLUMN name="billed_charges">
      <PATH syntax="XPath">/claims/claim/detail/billed_charges</PATH>
      <TYPE>character</TYPE>
      <DATATYPE>string</DATATYPE>
      <LENGTH>32</LENGTH>
    </COLUMN>
  </TABLE>
</SXLEMAP>
```

It should be noted that one of the reasons XML files become too complex for the default XML LIBNAME statement to handle is due to the relational (one-to-many / many-to-one) nature of the data being communicated in them. Therefore, it should not be surprising that the solution to this challenge is to explicitly define the data in terms of the separate "tables" of data within the file, as in the preceding example. Beginning with SAS 9.1, the XML Mapper tool became available to provide a user-friendly way of creating these .map files.

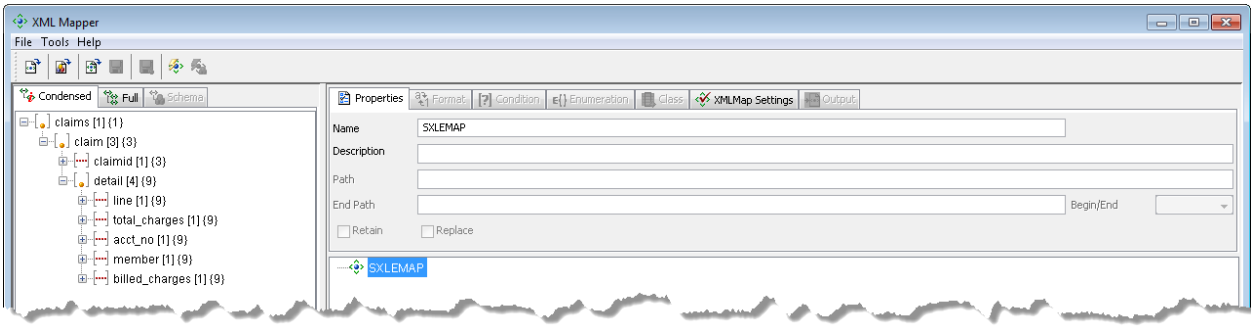
XML MAPPER

XML Mapper is a stand-alone Java™ applet that you can use to create and edit XML maps by (a) reading an XML source file, (b) reading an XML Schema document, or (c) editing an existing XML map file.

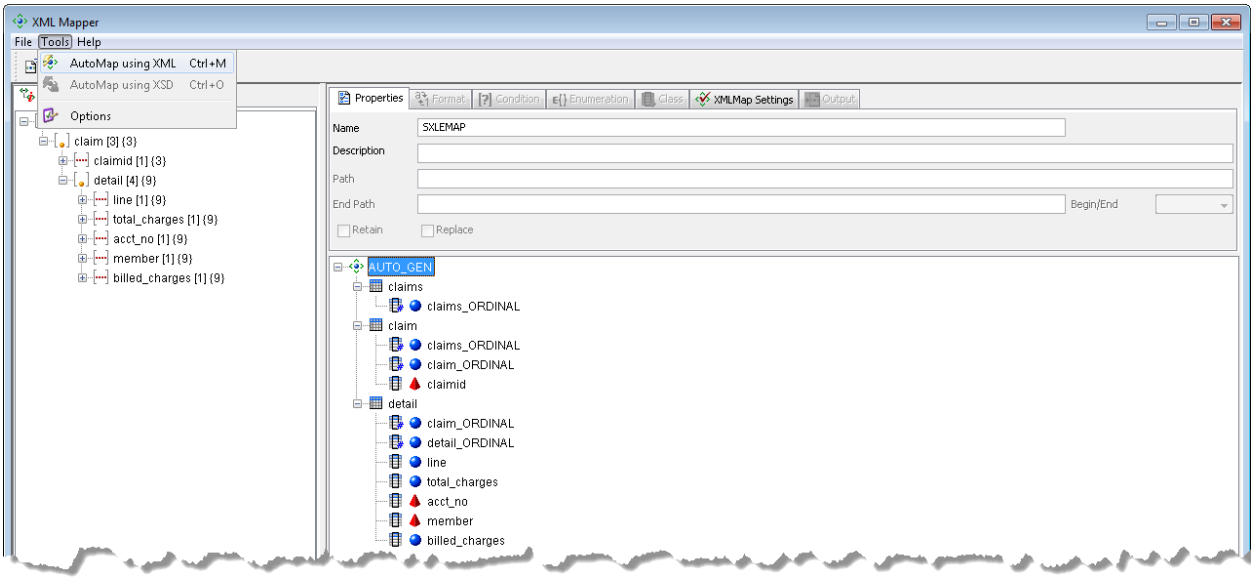
To read an XML source file with XML Mapper, simply choose "File►Open XML" from the menubar and navigate to the XML file you want to read.



Upon first opening an XML source file, you will notice that the levels of the hierarchy within the file are represented in the source file pane in the upper left of the interface.



In response to selecting the "AutoMap" function in the tools menu, XML Mapper attempts to generate an XML Map file, and presents a graphical depiction (in the "Properties" tab) of the SAS data tables that the map will parse from the XML file.

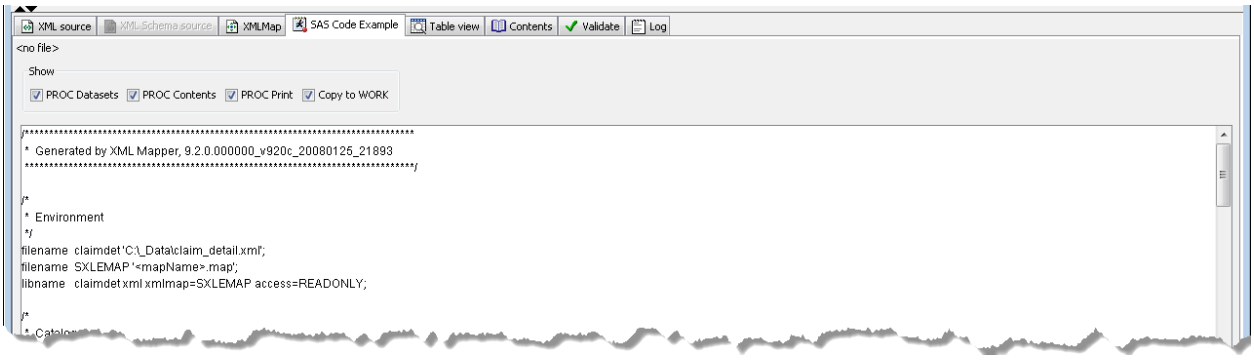


In addition to automatically creating a default representation of the datasets that can be created from the XML source data (in this case, "claims", "claim", and "detail"), XML Mapper is very good at making sure the hierarchical relationships identified in the data are maintained in a way that makes it simple to join data in tables that are subordinate to each other. This is done through the generation of surrogate primary and foreign key values (i.e., the "_ORDINAL" variables). Therefore, even though XML Mapper creates a number of separate tables in order to render all of the data in two dimensions, the hierarchical relationships between those tables are maintained. As Cox (2012) points out, "the automapping algorithm creates a representation of the XML that is as relational as possible from the available context. It does this, along with the creation of surrogate keys, so that the tables can be rejoined using PROC SQL."

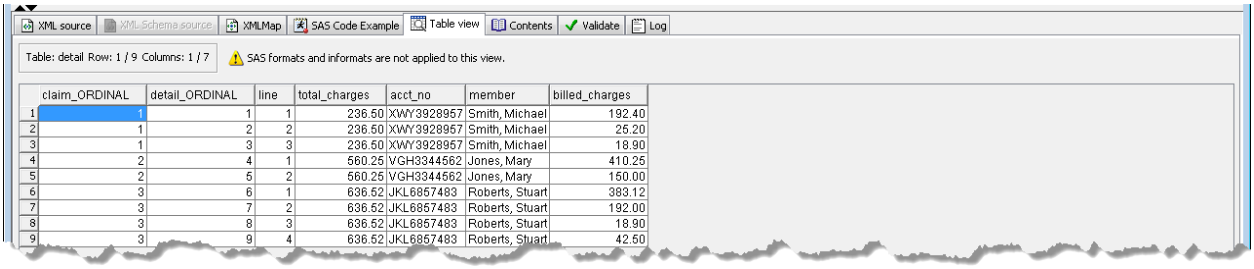
In addition to this graphical representation of the datasets that will be parsed from the XML file, the XML Mapper interface also shows you the XML Map file that is being generated ❶, example SAS code that can be used to interact with the SAS datasets made available by the .map file ❷, a view of the actual data in the dataset currently selected in the Map properties tab ❸, the names, data types, and lengths of the variables in that dataset (i.e., a PROC CONTENTS view of the datasets ❹, and a log of the activity that has transpired during your XML Mapper session ❺.

❶

❷



❸



	claim_ORDINAL	detail_ORDINAL	line	total_charges	acct_no	member	billed_charges
1	1	1	1	236.50	XWY3928957	Smith, Michael	192.40
2	1	2	2	236.50	XWY3928957	Smith, Michael	25.20
3	1	3	3	236.50	XWY3928957	Smith, Michael	18.90
4	2	4	1	560.25	VGH3344562	Jones, Mary	410.25
5	2	5	2	560.25	VGH3344562	Jones, Mary	150.00
6	3	6	1	636.52	JKL6857483	Roberts, Stuart	383.12
7	3	7	2	636.52	JKL6857483	Roberts, Stuart	192.00
8	3	8	3	636.52	JKL6857483	Roberts, Stuart	18.90
9	3	9	4	636.52	JKL6857483	Roberts, Stuart	42.50

❹

Name	Type	Length	Format	Informat	Label
claim_ORDINAL	numeric	8			
detail_ORDINAL	numeric	8			
line	numeric	8			
total_charges	numeric	8			
acct_no	character	10			
member	character	15			
billed_charges	numeric	8			

❺

	Description	Timestamp
	XML Mapper initialization: application mode	Tue Feb 04 23:33:29 CST 2014
	XML Mapper ready	Tue Feb 04 23:33:30 CST 2014
	Parsing with high validation.	Tue Feb 04 23:36:41 CST 2014
	XML file loaded: claim_detail.xml	Tue Feb 04 23:36:41 CST 2014
	Auto generated map	Tue Feb 04 23:39:31 CST 2014

After saving the map file as "claims map1", you can read data from the XML files as if they are stored as three datasets in the "clmfile" library.

```
LIBNAME local 'c:\_data\local\';
LIBNAME clmfile XML '\\datasrvr\claim_detail.xml' XMLMAP='c:\claims map1.map';

DATA local.claims;
 SET clmfile.claims;
RUN;
DATA local.claims;
 SET clmfile.claims;
RUN;
DATA local.detail;
 SET clmfile.detail;
RUN;
```

VIEWTABLE: Local.Claims

	claims_ORDINAL
1	1

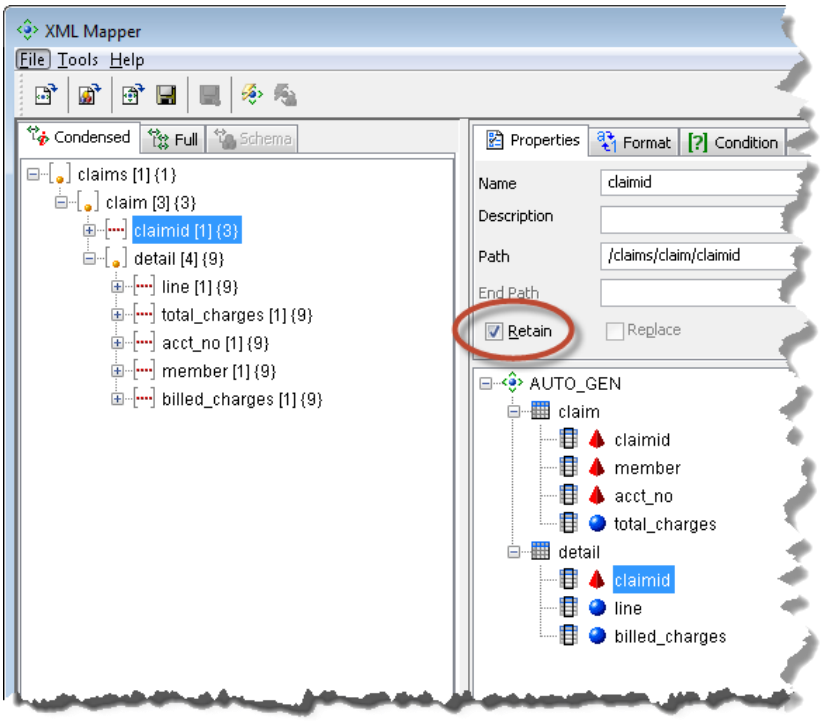
VIEWTABLE: Local.Claim

	claims_ORDINAL	claim_ORDINAL	claimid
1	1	1	584723988U
2	1	2	587439204T
3	1	3	866978852B

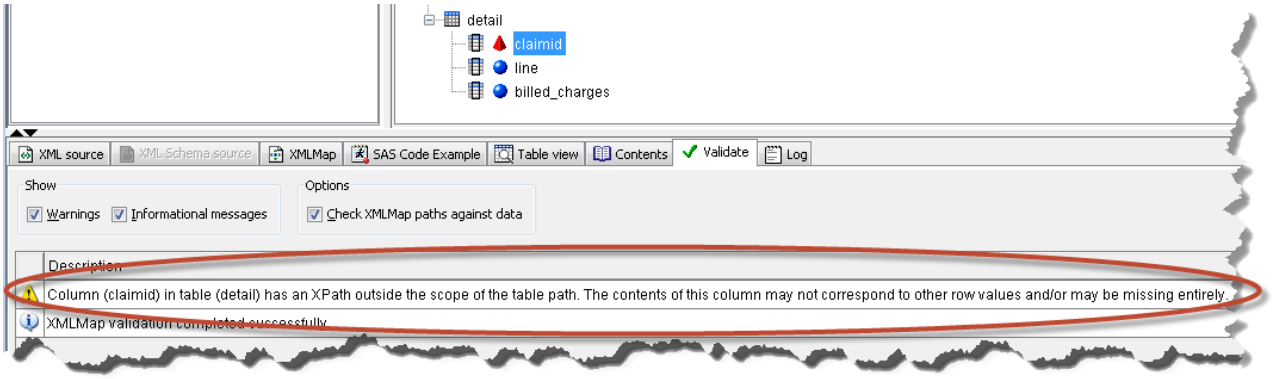
VIEWTABLE: Local.Detail

	claim_ORDINAL	detail_ORDINAL	line	total_charges	acct_no	member	billed_charges
1	1	1	1	236.5	XWY3928957	Smith, Michael	192.4
2	1	2	2	236.5	XWY3928957	Smith, Michael	25.2
3	1	3	3	236.5	XWY3928957	Smith, Michael	18.9
4	2	4	1	560.25	VGH3344562	Jones, Mary	410.25
5	2	5	2	560.25	VGH3344562	Jones, Mary	150
6	3	6	1	636.52	JKL6857483	Roberts, Stuart	383.12
7	3	7	2	636.52	JKL6857483	Roberts, Stuart	192
8	3	8	3	636.52	JKL6857483	Roberts, Stuart	18.9
9	3	9		636.52	JKL6857483	Roberts, Stuart	

In addition to using the auto-generated map files, however, you can also work with the XML Mapper interactively to add or remove elements to or from the tables represented in the table structure of the default map. Note that in the preceding example, the auto-generated map depicts a "claims" dataset that contains only the field "claims_ORDINAL". This "claims" dataset will contain only one record with a single variable and serves, in this example, only as a representation of the root tag. Because it will contain no actual data, it will be removed from the model by right-mouse-clicking on it and selecting delete. Also, the default mapping of the claims data resulted in several characteristics of the overall claim (e.g., account_no, total_charges, and member) being assigned to the "detail" dataset. Because these elements really describe the "claim" as a whole, they will be moved to the claim dataset by deleting them from the "detail" table and then dragging the XML elements in the source file pane on the left to the "claim" table representation in the XML Map Properties pane. As this is done, the XML map is automatically rewritten to reflect our new designation of these fields as belonging to the "claim" dataset. Finally, the "_ORDINAL"s from the "claim" and "detail" tables in the XML Map are deleted because the "claimid" forms a natural primary/foreign key relationship between these data elements. Finally, the "claimid" is dragged onto the "detail" table to complete the link between the "claim" and "detail" tables. The new map representation of the claims XML file is shown below. The new map file will parse two datasets from the claims XML file. Note, the XML file has not changed at all, but the changes to the map will change how data are extracted from that file structure.



When explicitly defining the relationship between the claim and the detail in this way, however, two things will happen in XML Mapper. First, as seen in the preceding figure, the "claimid" element in the detail table will automatically be marked as "retained". Second, validation of the map will generate the following warning.



This warning occurs because in the XML file, the "claimid" tag does not encompass the other elements that make up a "claim", so in order to assign this value to every row in the "detail" table each claim's "claimid" value will need to be retained as each individual detail element within the claim is read.

```
<?xml version="1.0" encoding="WINDOWS-1252"?>
<!-- XML File for Submission of Claims Data -->
- <claims xmlns="http://www.cdms-llc.com/ns/sasclaims/1.0">
  - <claim>
      <claimid>584723988U</claimid>
    - <detail>
        <total_charges>236.50</total_charges>
        <line>1</line>
        <acct_no>XWY3928957</acct_no>
        <member>Smith, Michael</member>
        <billed_charges>192.40</billed_charges>
      </detail>
    - <detail>
        <total_charges>236.50</total_charges>
        <line>2</line>
        <acct_no>XWY3928957</acct_no>
        <member>Smith, Michael</member>
        <billed_charges> 25.20</billed_charges>
      </detail>
    - <detail>
        <total_charges>236.50</total_charges>
        <line>3</line>
        <acct_no>XWY3928957</acct_no>
        <member>Smith, Michael</member>
        <billed_charges> 18.90</billed_charges>
      </detail>
    </claim>
```

Using this map to extract the SAS datasets from the claims file yields the following "detail" dataset.

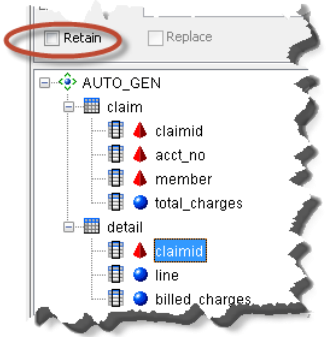
```
LIBNAME clmfile XML '\\datasrvr\claim_detail.xml' XMLMAP='c:\claims map2.map';

DATA local.detail;
 SET clmfile.detail;
RUN;
```

VIEWTABLE: Local.Detail

	claimid	line	billed_charges
1	584723988U	1	192.4
2	584723988U	2	25.2
3	584723988U	3	18.9
4	587439204T	1	410.25
5	587439204T	2	150
6	866978852B	1	383.12
7	866978852B	2	192
8	866978852B	3	18.9
9	866978852B		

If, on the other hand, the "claimid" was not retained in the map definition, the following dataset would be produced.

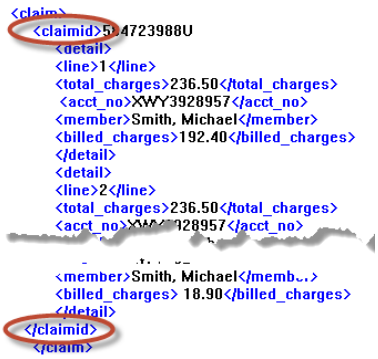
```
LIBNAME clmfile XML '\\datasrvr\claim_detail.xml' XMLMAP='c:\claims map3.map';

DATA local.detail;
 SET clmfile.detail;
RUN;
```

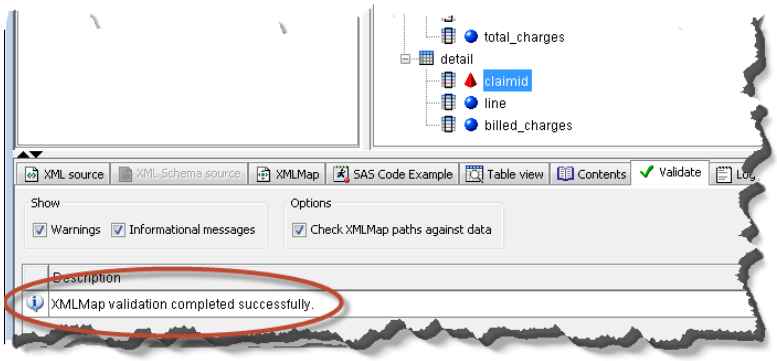


VIEWTABLE: Local.Detail

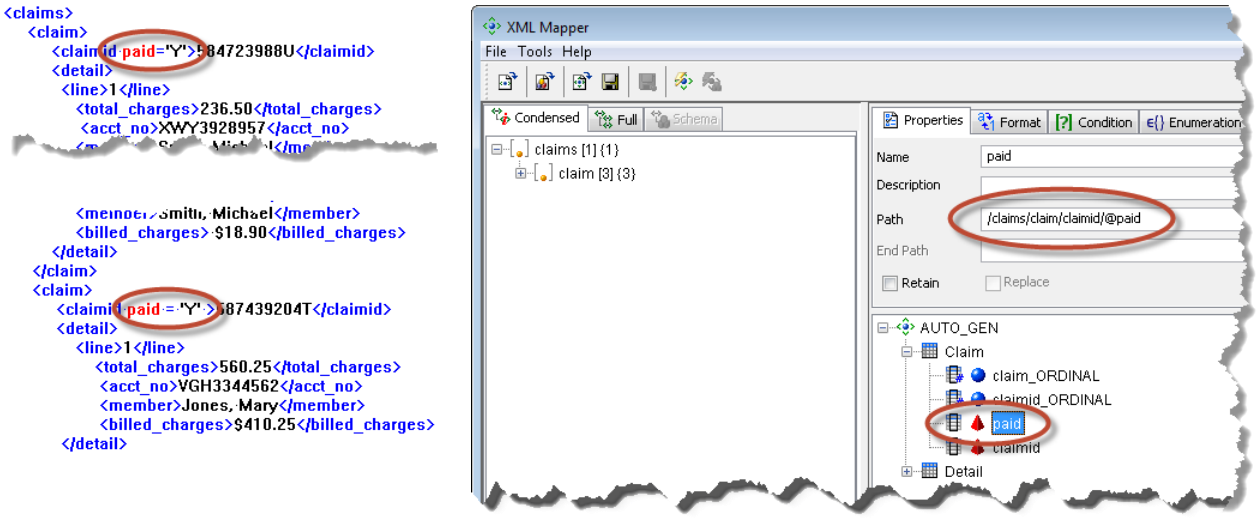
	claimid	line	billed_charges
1	584723988U	1	192.4
2		2	25.2
3		3	18.9
4	587439204T	1	410.25
5		2	150
6	866978852B	1	383.12
7		2	192
8		3	18.9
9			42.5

Finally, if the XML file's structure is such that the "claimid" tag does encapsulate the rest of the claim details with which it is associated, then adding the claimid to the detail table does not generate a warning when validating the map.

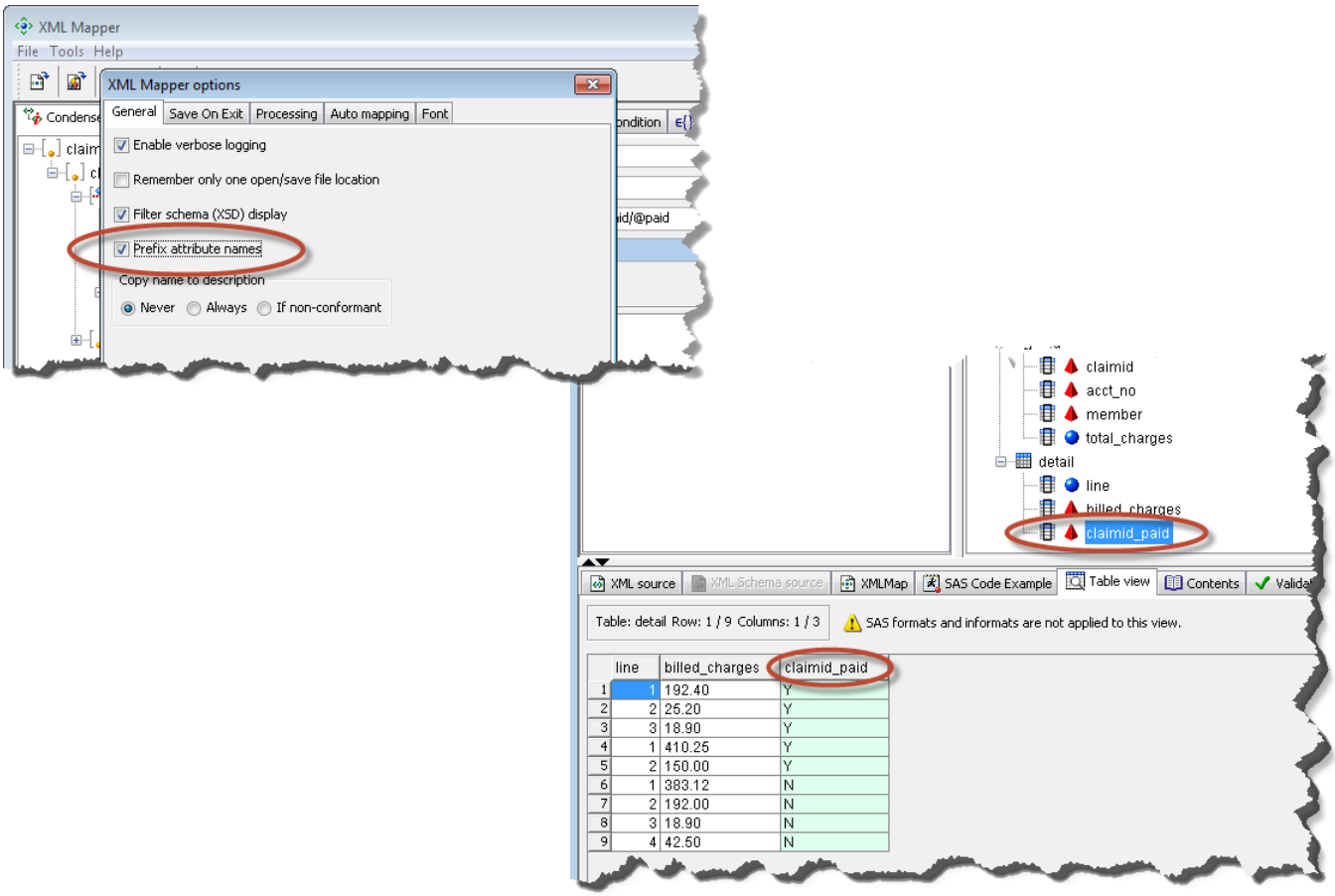




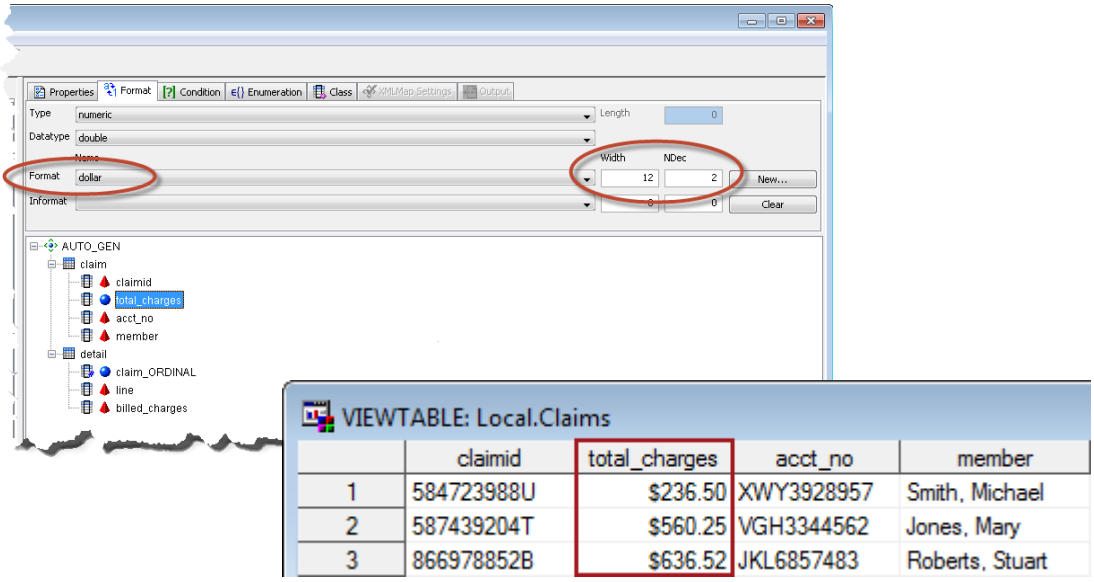
In addition to XML files comprised solely of XML "elements", the XML Mapper also recognizes "attributes" of XML elements and is capable of automatically adding them to the XML Map. In the following example, a claims file with a "paid" attribute is automapped from the XML file. Notice in the XML Map Properties that the path for the attribute is slightly different than the path for elements—ending with the XPATH syntax "@" to describe it as an attribute of the "claimid" element. Other than this difference in syntax, attributes are parsed from the XML file in the same fashion as elements.



Because this "paid" indicator is an attribute of the "claimid" it is tightly bound to the claimid element. Therefore, if this attribute were included in the "Detail" dataset, the XML Mapper has an option that allows you to make this relationship between "paid" and "claimid" explicit in the dataset. By selecting the "Prefix Attributes" option in the XML Mapper Options (Tools►Options), you can automatically add the attribute's element name to the beginning of the SAS variable name.

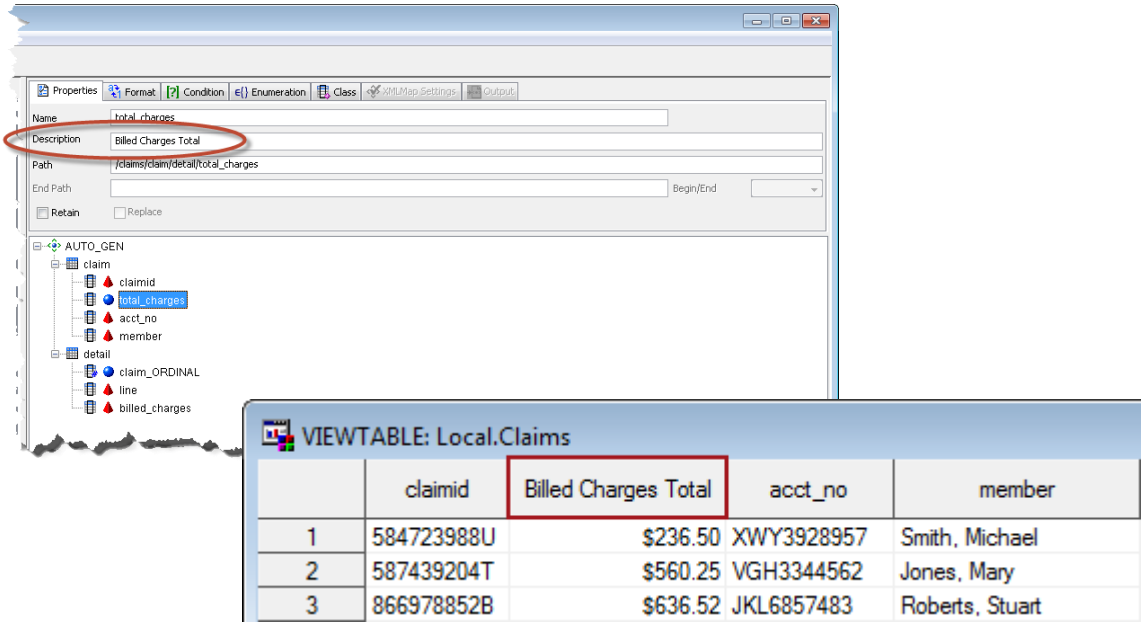


In addition to giving you the ability to manipulate the structure of the datasets extracted from your XML file, the XML Mapper provides you with a number of additional ways to customize your XML Map. As shown below, you can assign SAS FORMATS and INFORMATS to control the presentation of data in the resulting datasets and the transformation of data as they are read in from the XML file. For example, if you want to format the numeric element "total_charges" with a DOLLAR format, simply select the data element in the map, navigate to the "Format" tab, and select the desired DOLLAR format. As data are parsed and transformed into a SAS dataset, the format will be applied.



	claimid	total_charges	acct_no	member
1	584723988U	$236.50	XWY3928957	Smith, Michael
2	587439204T	$560.25	VGH3344562	Jones, Mary
3	866978852B	$636.52	JKL6857483	Roberts, Stuart

Similarly, you can provide variable LABELS (like "Billed Charges Total" in the following example) or rename the variables or dataset names from the XML element names by typing the desired label in the appropriate field on the Properties tab.

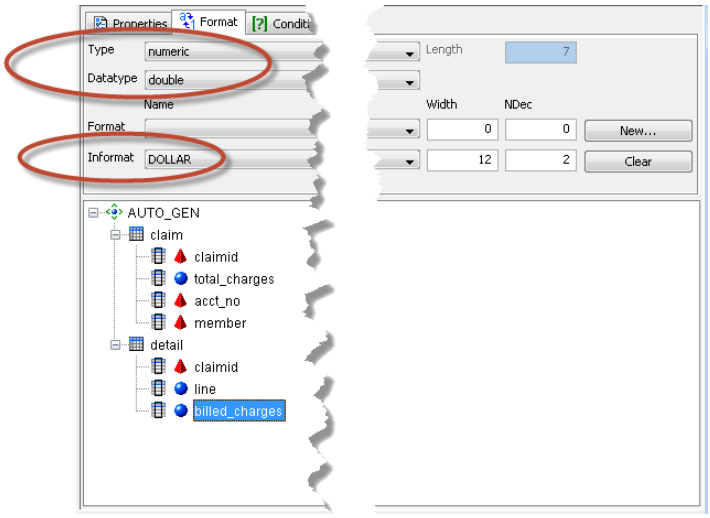


	claimid	Billed Charges Total	acct_no	member
1	584723988U	$236.50	XWY3928957	Smith, Michael
2	587439204T	$560.25	VGH3344562	Jones, Mary
3	866978852B	$636.52	JKL6857483	Roberts, Stuart

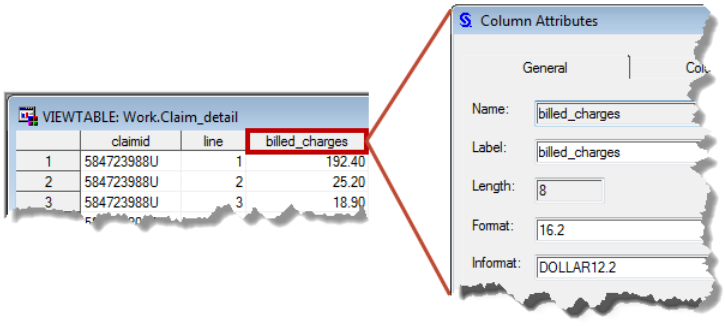
In addition to formatting the SAS dataset columns resulting from reading data through the XML Map, the map can also be configured to convert data as it is read in from the XML file. For example, an auto-generated map based on the adjacent XML file would identify the "billed_charges" element (and corresponding SAS variable) as a character string. It is more than likely, however, that if you are going to read in dollar values, you will want to perform arithmetic operations on them.

```
<?xml version="1.0" encoding="windows-1252"?>
<!-- XML File for Submission of Claims Data -->
<claims>
   <claim>
      <claimid>584723988U</claimid>
         <detail>
         <line>1</line>
         <total_charges>236.50</total_charges>
          <acct_no>XWY3928957</acct_no>
         <member>Smith, Michael</member>
         <billed_charges>$192.40</billed_charges>
         </detail>
         <detail>
         <line>2</line>
         <total_charges>236.50</total_charges>
         <acct_no>XWY3928957</acct_no>
         <member>Smith, Michael</member>
         <billed_charges> $25.20</billed_charges>
         </detail>
```

In order convert the data to numeric values as the data are read from the XML file, you can change the data type of the SAS variable to "Numeric" and specify the precision as "double" (rather than integer). Next specify the INFORMAT as DOLLAR. Just as the FORMAT is an instruction-set that tells SAS how to "display" data, the INFORMAT is an instruction-set that tells SAS how to "read" data. In this case, the instruction to SAS is to ignore the dollar sign ($) and thousand-separating commas in reading what is, otherwise, a numeric variable. With this INFORMAT applied, the data in "billed_charges" will now be correctly interpreted by the XML Libname Engine as a numeric SAS data element.

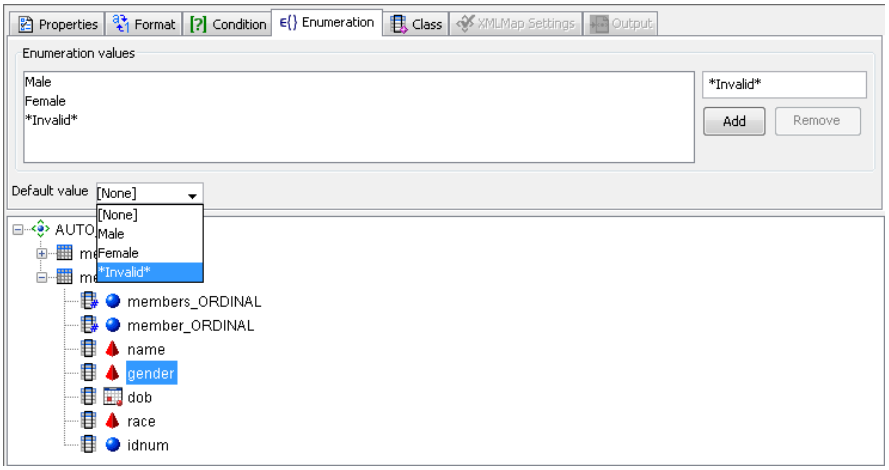



```
LIBNAME clmfile XML '\\datasrvr\claim_detail.xml' XMLMAP='c:\dollar.MAP';

DATA work.claim_detail;
 SET clmfile.detail;
RUN;
```



Another way in which XML Mapper can help control and manipulate data read from an XML file is with enumeration. An enumeration is one of several types of "facets" (or, restrictions) that can be defined in an XML schema. Enumerations limit the valid values a specific element can take. In the following example, the "members" XML file will be mapped with XML Mapper. After opening the file and auto-generating a default map, the "gender" variable is selected in the map representation of the "member" dataset. Next the values "Male", "Female", and "*Invalid*" are entered as enumeration values and the value "*Invalid*" is selected as the default value. Similarly, valid values are entered for "race", and the map is saved as "members.map".

```
<?xml version="1.0" encoding="windows-125
<members>
  <member>
    <name>Smith, Michael</name>
    <gender>Male</gender>
    <dob>06/13/1982</dob>
    <race>C</race>
    <idnum>47584734</idnum>
  </member>
  <member>
    <name>Ramos, Miguel</name>
    <gender>Hispanic</gender>
    <dob>11/10/1981</dob>
    <race>Male</race>
    <idnum>48566822</idnum>
  </member>
  <member>
    <name>Jones, Sara</name>
    <gender>Female</gender>
    <dob>03/02/1968</dob>
    <race>A</race>
    <idnum>93755823</idnum>
  </member>
</members>
```



When "members.map" is used to read the "members" dataset from the members.xml file, you can see that the invalid values in the dataset are replaced with the default value—which in this case serves as a flag for invalid values that will need to be investigated further to determine why invalid values are being written to the source file.

```
LIBNAME members XML '\\datasrvr\claim_detail.xml' XMLMAP='C:\members.map';

DATA local.members;
 SET members.member;
RUN;
```

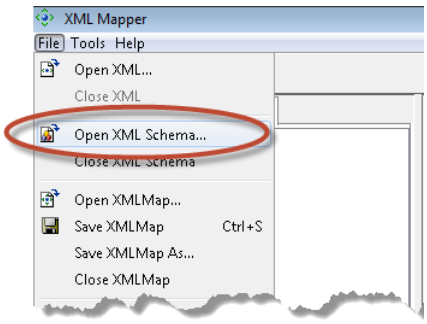
VIEWTABLE: Local.Members

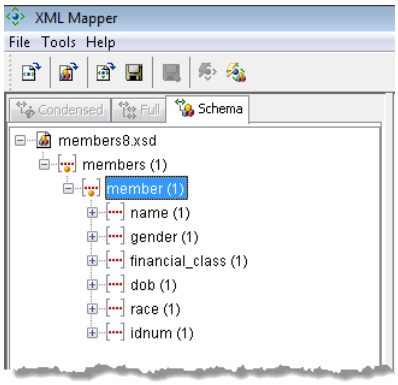
	members_ORDINAL	member_ORDINAL	name	gender	dob	race	idnum
1	1	1	Smith, Michael	Male	1982-06-13	C	47584734
2	1	2	Ramos, Miguel	*Invalid*	1981-11-10	*	48566822
3	1	3	Jones, Sara	Female	1968-03-02	A	93755823

Enumeration may also be specified in an XML Schema Definition (XSD) file. Neither the current discussion of XML nor its predecessor (Schacherer, 2013) has addressed XSD files, but they are an important part of understanding XML. The XML Schema Definition Language (XSDL) is comprised of XML elements and attributes used to define the namespaces, elements, and attributes that form a valid representation of a schema. In the example to the right, the schema definition for the "members" schema is presented. Note that in addition to specifying an enumeration restriction for the valid values of the "gender" variable ❶, the XSD file also specifies the maximum allowable length of the "name" variable as 100 characters ❷ and there is another variable, "financial_class" that is specified as a valid element in XML files in this schema ❸. Although the intended use of an XSD file is to validate an XML file that is purportedly an example of that schema, the SAS XML LIBNAME Engine, unfortunately, does not validate data against the XSD—assuming, instead, that well-formed XML meets the qualifications of any applicable XSD file.

```
<?xml version="1.0"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="members">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="member">
          <xs:complexType>
            <xs:sequence>
              <xs:element name="name">
              <xs:simpleType>
                <xs:restriction base="xs:string">
                <xs:length value="100"/>
                </xs:restriction>
              </xs:simpleType>
              </xs:element>
              <xs:element name="gender">
              <xs:simpleType>
                <xs:restriction base="xs:string">
                  <xs:enumeration value="Male"/>
                  <xs:enumeration value="Female"/>
                </xs:restriction>
              </xs:simpleType>
              </xs:element>
              <xs:element name="financial_class" type="xs:string"/>
              <xs:element name="dob" type="xs:string"/>
              <xs:element name="race" type="xs:string"/>
              <xs:element name="idnum" type="xs:string"/>
            </xs:sequence>
          </xs:complexType>
        </xs:element>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

However, if an XSD file is available for the XML file you will be importing, it is a good idea to use that XSD file as the basis for your XML Map instead of using a sample XML file. To build an XML Map based on an XSD file, simply select "Open XML Schema" from the File menu, navigate to the .XSD file, and choose "Open". The representation of a valid file within this XML schema is then presented in the "Schema" tab.



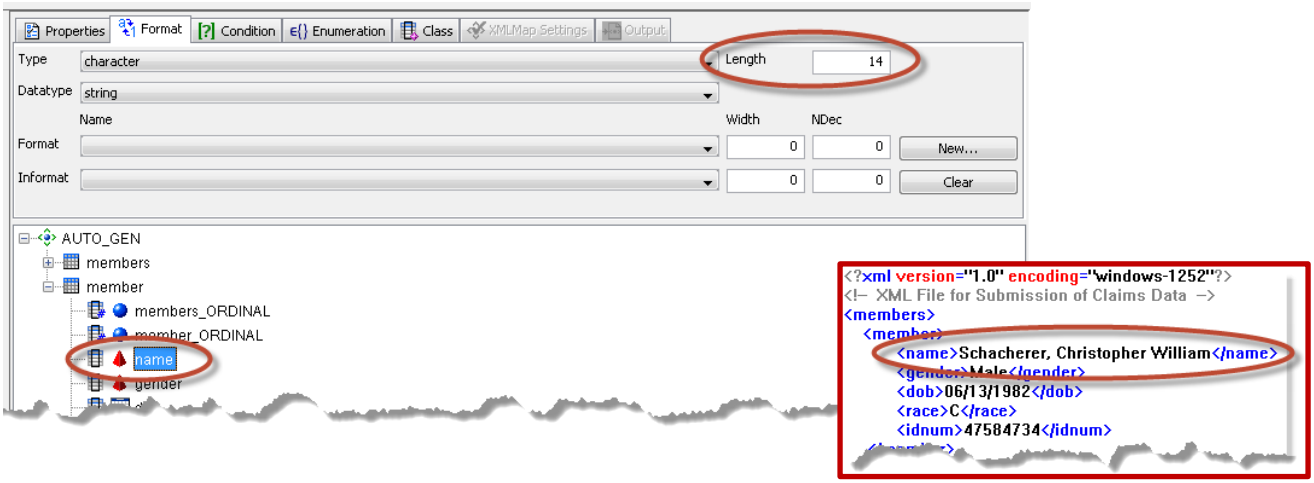


Automapping the XSD file works the same way as automapping an XML file, but the main advantage in using the XSD file is that your map will be guaranteed to have the same variables and variable characteristics as in the schema definition. In the following example, the xml map generated from the previously received "members.xml" file is used to import a subsequently version of the members file—perhaps as part of a monthly data processing job.

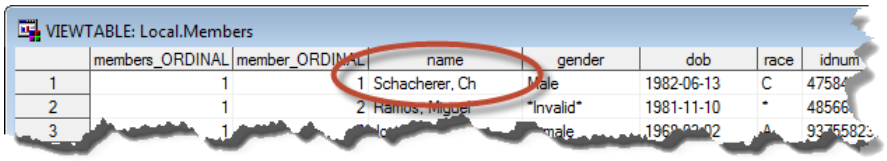
```
LIBNAME members XML '\\datasrvr\claim_detail.xml' XMLMAP='c:\members.map';

DATA local.members;
 SET members.member;
RUN;
```

```
413   LIBNAME members XML 'c:\_data\members2.xml' XMLMAP='c:\_Data\members.map';
NOTE: Libref MEMBERS was successfully assigned as follows:
      Engine:        XML
      Physical Name: c:\_data\members2.xml
414
415   DATA local.members;
416    SET members.member;
417   RUN;

WARNING: At least one data value for column name may have been truncated. Maximum
length of data encountered during input was 31.
```

As described in the preceding log, there was a potential misstep in reading this "members.xml" file. In looking at the map file produced from the previous XML file, you see that the maximum length for "name" was 14 characters, but in the "members.xml" file being read in the preceding example, one member has a name that is 31 characters.

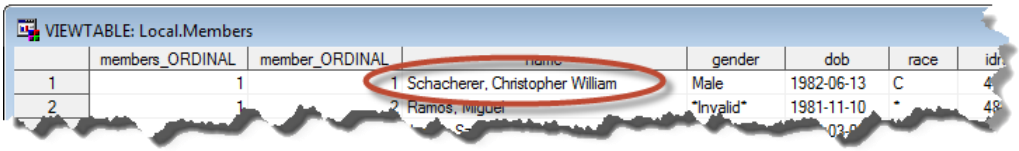


Looking back at the "members.xsd" file, however, you can see that a 31-character "name" is completely acceptable within the "members" schema, so the "members.xml" file is a valid example of the "members" schema. The problem is that the xml file on which "members.map" was originally built only contained "name" elements with a maximum length of 14 characters. As a result, subsequent, valid exemplars of the "members" schema are read incorrectly by your SAS program.

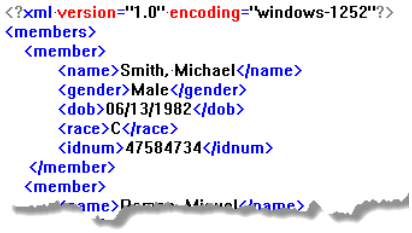


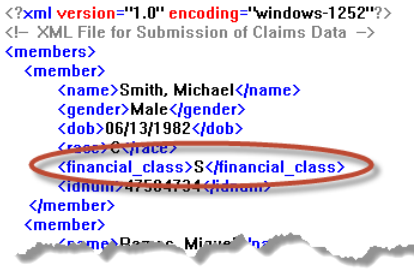
Basing the "members.map" file, on the .XSD file correctly indicates the maximum valid length of the name variable as 100, and reading the same XML file with this new XSD-based map accurately reads the data from the XML file.

```
LIBNAME members XML '\\datasrvr\claim_detail.xml' XMLMAP='c:\members-XSD.map';

DATA local.members;
 SET members.member;
RUN;
```



Another thing to consider when basing your map on an XML file rather than on an XSD file is whether subsequent versions of the XML source file might contain elements that are not present in the XML file on which you based your map. In the following example, the XML file (left) on which the original "members.map" was based contains no occurrences of the XML element "financial_class". However, "financial_class" is a valid element in the schema defined by members.xsd. Using the map based on the XML file on the left, you may never even know that "financial_class" was a valid, available variable under this schema.





Even when a file containing "financial class" is received, reading the data through the original XML Map file (created based on the XML example without any "financial_class" tags), would give you no clue as to the occurrence of this extra variable. Reading the data with the existing map (which contains no reference to the element "financial_class") would produce no errors or warnings because you have simply not included "financial_class" as a variable that you want to include from the source file.

```
LIBNAME members XML '\\datasrvr\claim_detail.xml' XMLMAP= 'C:\members.map';

DATA local.members;
 SET members.member;
RUN;
```

VIEWTABLE: Local.Members

	members_ORDINAL	member_ORDINAL	name	gender	dob	race	idnum
1	1	1	Smith, Michael	Male	1982-06-13	C	47584734
2	1	2	Ramos, Miguel	*Invalid*	1981-11-10	*	48566822
3	1	3	Jones, Sara	Female	1968-03-02	A	93755823

But reading the exact same XML file with the XSD-based map correctly reads in the "financial_class" variable.

```
LIBNAME members XML 'c:\_data\finclass.xml' XMLMAP= 'C:\members-xsd.map';

DATA local.members;
 SET members.member;
RUN;
```

VIEWTABLE: Local.Members

	members_ORDINAL	member_ORDINAL	name	gender	financial_class	dob	race	idnum
1	1	1	Smith, Michael	Male	S	06/13/1982	C	47584734
2	1	2	Ramos, Miguel	*Invalid*		11/10/1981	*	48566822
3	1	3	Jones, Sara	Female		03/02/1968	A	93755823

Given the role played by the XSD file in defining the schemas of which your XML source file is but one example, you are well served by creating your XML Maps based on the XSD file when one is available to you.

GENERATING XML FILES FROM SAS DATASETS

Most of the examples so far have focused on reading XML files, but it is equally important to know how to write XML files from a SAS dataset. So far, the sole example of doing this was based on a relatively simple data structure. In this case the rectangular structure of the source file that was read into SAS was readily reproduced when the data were written back out to an XML file. By defining the library "claimout" as an XML library pointing to the file "2012-07.xml", the execution of a simple DATA step writes the well-formed, rectangular XML file depicted below.

```
LIBNAME claimout XML 'c:\2012-07.xml';
DATA claimout.claim_detail;
 SET work.claims;
RUN;
```

VIEWTABLE: Work.Claims

	claimid	total_charges	line	acct_no	member	billed_charges
1	584723988U	236.50	1	XWY3928957	Smith, Michael	192.40
2	584723988U	236.50	2	XWY3928957	Smith, Michael	25.20
3	584723988U	236.50	3	XWY3928957	Smith, Michael	18.90
4	587439204T	560.25	1	VGH3344562	Jones, Mary	410.25
5	587439204T	560.25	2	VGH3344562	Jones, Mary	150.00

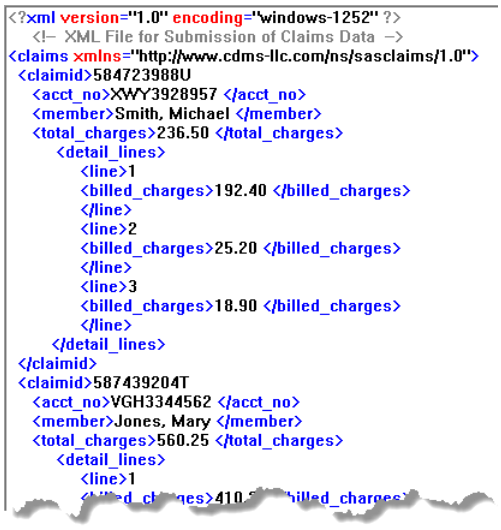
```
<?xml version="1.0" encoding="WINDOWS-1252"?>
- <TABLE>
 - <CLAIM_DETAIL>
     <claimid> 584723988U </claimid>
     <total_charges> 236.50 </total_charges>
     <line> 1 </line>
     <acct_no> XWY3928957 </acct_no>
     <member> Smith, Michael </member>
     <billed_charges> 192.40 </billed_charges>
   </CLAIM_DETAIL>
 - <CLAIM_DETAIL>
     <claimid> 584723988U </claimid>
     <total_charges> 236.50 </total_charges>
     <line> 2 </line>
     <acct_no> XWY3928957 </acct_no>
     <member> Smith, Michael </member>
     <billed_charges> 25.20 </billed_charges>
   </CLAIM_DETAIL>
 - <CLAIM_DETAIL>
     <claimid> 584723988U </claimid>
     <total_charges> 236.50 </total_charges>
     <line> 3 </line>
     <acct_no> XWY3928957 </acct_no>
     <member> Smith, Michael </member>
     <billed_charges> 18.90 </billed_charges>
   </CLAIM_DETAIL>
 - <CLAIM_DETAIL>
     <claimid> 587439204T </claimid>
     <total_charges> 560.25 </total_charges>
     <line> 1 </line>
```

By default, when writing a SAS dataset out to an XML file in this way, each record in the dataset is output as a set of child tags subordinate to a tag with a name corresponding to the name of the target dataset—in this case, "CLAIM_DETAIL"—and as was discussed earlier, the default root element is always "TABLE". Control over the structure of the XML file is relatively limited in this approach. Of course, you could write the xml file and then replace the root tag by editing the file in a text editor, but that requires a manual step to be introduced into the process and this approach is not always available—as when you want to write your file directly to an FTP directory. Additionally,

the default rectangular structure severely limits the XML structures that can be produced by exporting data through the XML Libname Engine. One way of having more direct control over the creation of your XML files is to borrow a technique that was popular early in the adoption of XML by SAS programmers. In the following example, this method—which uses simple "PUT" and "FILENAME" statements—is used to generate a more sophisticated structure for the claims and claim details data used in the previous examples. After sorting the data by "claimid" and "line", a DATA step is used to control the tagging of the data elements to produce one of the many possible XML structures that can be written from the "claims" dataset.

```
FILENAME claims 'c:\_data\claims put.xml' ;

DATA _NULL_;
FILE claims;
SET work.claims end=eof_claims;
    BY claimid line;
 IF _n_ = 1 THEN DO;
    PUT '<?xml version="1.0" encoding="windows-1252" ?>';
    PUT '     <!--  XML File for Submission of Claims Data  -->';
    PUT '     <claims xmlns="http://www.cdms-llc.com/ns/sasclaims/1.0">';
 END;
 IF first.claimid THEN DO;
    PUT '<claimid>'claimid;
    PUT '     <acct_no>' acct_no '</acct_no>';
    PUT '     <member>' member '</member>';
    PUT '     <total_charges>' total_charges '</total_charges>';
    PUT '          <detail_lines>';
 END;
 IF first.line THEN DO;
 PUT '               <line>' line ;
 PUT '               <billed_charges>' billed_charges '</billed_charges>';
 PUT '               </line>';
 END;
IF last.claimid THEN DO;
  PUT '          </detail_lines>';
  PUT '</claimid>';
  END;
IF eof_claims THEN
PUT '</claims>';
RUN;
```

Note that the resulting file has a deeper hierarchical structure than that produced from the default export through the XML Libname Engine—presenting "acct_no", "member", "total_charges" and "claimid" just once in association with each claim and specifying a different level of the hierarchy for subordinate detail records. In addition, using this method you have full control over the value of the root tag (rather than accepting the default value "TABLE", or making changes to the default value for the root tag).



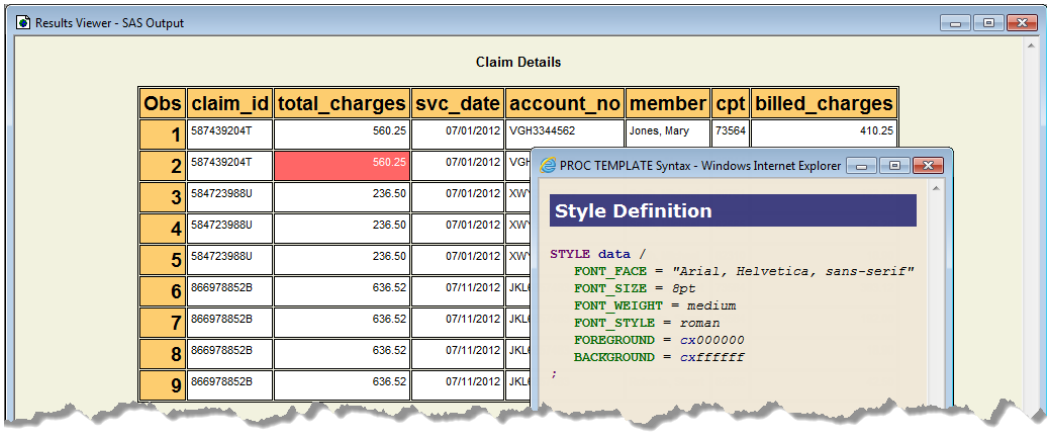
Although this DATA step approach to writing XML seems like a significant improvement over accepting the default rectangular form of the file produced by the LIBNAME engine, you also need to consider that as the desired structure of the XML becomes more complex, the programming required to produce the XML structure will increase in complexity also. In addition, producing XML files in this manner will also require that you either include this logic in each program from which you wish to produce files using this XML structure or create a macro that will handle the data manipulation and write the data out to the specified file.

A more robust means of writing XML files from SAS is to use the Output Delivery System's markup tagsets.

ODS MARKUP TAGSETS

The Output Delivery System (ODS) "enables you to produce SAS procedure and DATA step output to many different destinations" (SAS, 2011b, p. 33). Common uses of ODS are to write reports, listings, and analytic output to destinations such as HTML and PDF files with the main focus being on controlling the appearance of the output (for example, the font, spacing, and color of output elements). SAS accomplishes this feat in ODS by combining raw data generated by a PROC step with a standard structure for naming the output elements associated with that PROC (e.g., rowheader, header, data, etc.). These output elements (referred to collectively as the "Table Definition" for a given PROC) are used to identify the elements within a Style Definition that determine the appearance of the output. In the following example, the result of a PROC PRINT step involving the "claim_detail" dataset is sent via ODS to the HTML destination using the SAS style template "harvest". The HTML destination is opened with the ODS HTML statement that specifies the output HTML file and the style being applied to data in the table definition. After specifying a title for the output, the PROC PRINT step is executed and ODS applies the style definition to the table elements associated with the PROC PRINT output—resulting, for example, in the data elements being defined as font face/size "Arial 8" and the background color of the row and column headers being harvest gold.

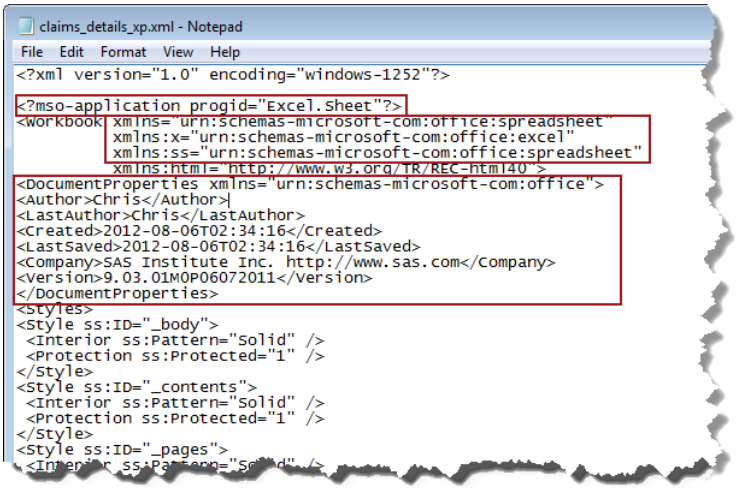
```
ODS HTML  FILE='c:\_data\claim_details.html' STYLE = HARVEST;

TITLE1 'Claim Details';
PROC print DATA=work.claim_detail;
RUN;

ODS HTML CLOSE;
```



Obs	claim_id	total_charges	svc_date	account_no	member	cpt	billed_charges
1	587439204T	560.25	07/01/2012	VGH3344562	Jones, Mary	73564	410.25
2	587439204T	560.25	07/01/2012	VGH			
3	584723988U	236.50	07/01/2012	XW			
4	584723988U	236.50	07/01/2012	XW			
5	584723988U	236.50	07/01/2012	XW			
6	866978852B	636.52	07/11/2012	JKL			
7	866978852B	636.52	07/11/2012	JKL			
8	866978852B	636.52	07/11/2012	JKL			
9	866978852B	636.52	07/11/2012	JKL			

In addition to HTML, PDF, RTF, and several other output destinations intended for use in rendering PROC and DATA step output to reader-friendly reporting formats, beginning in SAS 8.2 the MARKUP destination was added "to enable users to control the markup language tags or markup text written to their result files" (Haworth, Zender, & Burlew, 2009, p. 321). This control is exercised through application of different "tagsets"—or, instructions about which tags and associated attributes should be written to the output file. In the following example, the EXCELXP tagset is specified and the resulting file "claim_details_xp.xml" includes tags with namespace declarations and processing instructions necessary for Excel to process the data and open the file in Excel with formatting specified by the SAS default style definition.

```
ODS MARKUP TAGSET=EXCELXP FILE='c:\_data\claim_details_xp.xml';
PROC print DATA=work.claim_detail;
RUN;
ODS MARKUP CLOSE;
```

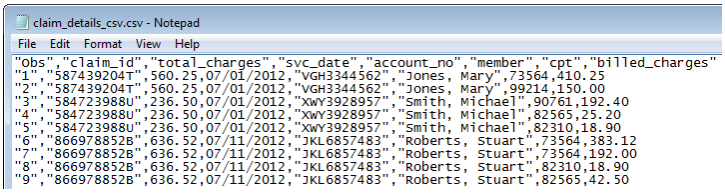


claims_details_xp.xml

	A	B	C	D	E	F	G	H
1	Obs	claim_id	total_charge	svc_date	account_	member	cpt	billed_charges
2	1	587439204T	560.25	07/01/2012	2	Jones, Mary	73564	410.25
3	2	587439204T	560.25	07/01/2012	2	Jones, Mary	99214	150
4	3	584723988U	236.5	07/01/2012	7	Smith, Michael	90761	192.4
5	4	584723988U	236.5	07/01/2012	7	Smith, Michael	82565	25.2
6	5	584723988U	236.5	07/01/2012	7	Smith, Michael	82310	18.9
7	6	866978852B	636.52	07/11/2012	JKL6857483	Roberts, Stuart	73564	383.12
8	7	866978852B	636.52	07/11/2012	JKL6857483	Roberts, Stuart	73564	192
9	8	866978852B	636.52	07/11/2012	JKL6857483	Roberts, Stuart	82310	18.9
10	9	866978852B	636.52	07/11/2012	JKL6857483	Roberts, Stuart	82565	42.5

Conversely, the following example of the CSV tagset creates output with almost no control syntax—save for the column names, comma delimiting of values, and quotation marks around string content.

```
ODS MARKUP TAGSET=CSV FILE='c:\_data\claim_details_csv.csv';
PROC PRINT DATA=work.claim_detail;
RUN;
ODS MARKUP CLOSE;
```



In both of the preceding examples, outputting the same SAS dataset with different tagsets results in very different files. In much the same way as style definitions can be applied to data elements to impact the appearance of the output generated by ODS, within the MARKUP destination different "tagsets" can also be applied to influence the markup tags written to the ODS output file. Also, like style definitions, tagsets are event-driven—responding to different triggering events found in the data upon which ODS is acting. Take a look at the events that ODS is evaluating when a simple PROC PRINT is issued against the "claim_detail" dataset.

```
ODS MARKUP TAGSET=EVENT_MAP FILE='c:\_data\claim_detail_events.xml';
PROC PRINT DATA=work.claim_detail;
RUN;
ODS MARKUP CLOSE;
```

```
<?xml version="1.0" encoding="windows-1252" ?>
- <doc operator="Chris" sasversion="9.3" saslongversion="9.03.01M0P06072011" date="2012-08-06" time="03:09:08" encoding="windows-
  1252" event_name="doc" trigger_name="attr_out" class="body" just="l">
  - <doc_head event_name="doc_head" trigger_name="attr_out" class="body" just="l">
      <doc_meta event_name="doc_meta" trigger_name="attr_out" class="body" just="l" />
      <auth_oper event_name="auth_oper" trigger_name="attr_out" class="body" just="l" />
      <doc_title event_name="doc_title" trigger_name="attr_out" class="body" just="l" />
      <stylesheet_link event_name="stylesheet_link" trigger_name="attr_out" just="c" />
    - <javascript event_name="javascript" trigger_name="attr_out" class="body" just="l">
        <startup_function event_name="startup_function" trigger_name="attr_out" class="startupfunction" just="l" />
        <shutdown_function event_name="shutdown_function" trigger_name="attr_out" class="shutdownfunction" just="l" />
      </javascript>
    </doc_head>
  - <doc_body event_name="doc_body" trigger_name="attr_out" class="body" just="c">
    - <proc event_name="proc" trigger_name="attr_out" name="Print" just="c">
        <anchor event_name="anchor" trigger_name="attr_out" class="body" name="IDX" index="IDX" just="c" />
      - <page_setup event_name="page_setup" trigger_name="attr_out" class="body" index="IDX" just="c">
```

When the PROC PRINT step was executed, all of these events (and many, many more) were interpreted by ODS to determine (based on the specified tagset) how the markup syntax should be written for this particular output file. In this case, for example, the "doc" event is associated with several data points that the "EVENT_MAP" tagset instructs ODS to write out as attributes of the "doc" element. In the previous EXCELXP example the tagset instruction with respect to the "doc" event is, in part, to write a separate "<version>" tag containing the same content as the "saslongversion" tag attribute in the current example. The same events and content are being processed in both examples; the only difference is in the instructions the different tagsets provide for responding to those events.

Understanding that tagsets are simply instructions for determining which tags to write to the output file (and how to write them), it stands to reason that if you could write your own tagset you would have complete control over how a markup file was produced from a source dataset. With PROC TEMPLATE you can do exactly that. PROC TEMPLATE gives you the ability to write both markup tagsets and style templates. Although an in-depth discussion of PROC TEMPLATE is outside the scope of this paper, a brief example, based on the "claim_detail" dataset is provided below.

The first step in this process has already been taken; PROC PRINT was run with the EVENT_MAP tagset to identify the events we can expect to encounter when writing out data with our soon-to-be-created custom tagset. In addition to the previously discussed "doc" event, you can see in the following section of the event map data that the events associated with the data elements we want to write out are found in the <data> tag, which is a child of the <row> tag. In order for a custom tagset to output tags containing these values the tagset will need to define what happens when these events (and a few others) are encountered.

```
- <table_body event_name="table_body" trigger_name="attr_out" output_name="Print" output_label="Data Set
    WORK.CLAIM_DETAIL" colcount="1" index="IDX" just="c">
  - <row event_name="row" trigger_name="attr_out" output_name="Print" output_label="Data Set
      WORK.CLAIM_DETAIL" section="body" colcount="1" index="IDX" just="c">
    <header event_name="header" trigger_name="attr_out" output_name="Print" output_label="Data Set
      WORK.CLAIM_DETAIL" section="body" class="rowheader" value="1" row="2" data_row="1"
      colcount="1" col="1" col_id="1" name="Obs" label="Obs" type="double" rawvalue="P/AAAAAAAAA="
      index="IDX" just="r" colwidth="8" scale="0" precision="0" />
    <data event_name="data" trigger_name="attr_out" output_name="Print" output_label="Data Set
      WORK.CLAIM_DETAIL" section="body" class="data" value="587439204" row="2" data_row="1"
      colcount="1" col="2" col_id="2" name="claim_id" label="claim_id" type="string" index="IDX" just="l"
      colwidth="10" scale="0" precision="0" />
    <data event_name="data" trigger_name="attr_out" output_name="Print" output_label="Data Set
      WORK.CLAIM_DETAIL" section="body" class="data" value="560.25" row="2" data_row="1"
      colcount="1" col="3" col_id="3" name="total_charges" label="total_charges" type="double"
      rawvalue="QIGCAAAAAAA=" index="IDX" just="r" colwidth="8" scale="0" precision="0" />
    <data event_name="data" trigger_name="attr_out" output_name="Print" output_label="Data Set
      WORK.CLAIM_DETAIL" section="body" class="data" value="07/01/2012" row="2" data_row="1"
      colcount="1" col="4" col_id="4" name="svc_date" label="svc_date" type="double"
      rawvalue="QNK5wAAAAAA=" index="IDX" just="r" colwidth="10" scale="10" precision="0" />
    <data event_name="data" trigger_name="attr_out" output_name="Print" output_label="Data Set
      WORK.CLAIM_DETAIL" section="body" class="data" value="VGH3344562" row="2" data_row="1"
      colcount="1" col="5" col_id="5" name="account_no" label="account_no" type="string" index="IDX"
      just="l" colwidth="10" scale="0" precision="0" />
    <data event_name="data" trigger_name="attr_out" output_name="Print" output_label="Data Set
      WORK.CLAIM_DETAIL" section="body" class="data" value="Jones, Mary" row="2" data_row="1"
      colcount="1" col="6" col_id="6" name="member" label="member" type="string" index="IDX" just="l"
      colwidth="18" scale="0" precision="0" />
    <data event_name="data" trigger_name="attr_out" output_name="Print" output_label="Data Set
      WORK.CLAIM_DETAIL" section="body" class="data" value="73564" row="2" data_row="1"
      colcount="1" col="7" col_id="7" name="cpt" label="cpt" type="string" index="IDX" just="l"
      colwidth="5" scale="0" precision="0" />
    <data event_name="data" trigger_name="attr_out" output_name="Print" output_label="Data Set
      WORK.CLAIM_DETAIL" section="body" class="data" value="410.25" row="2" data_row="1"
      colcount="1" col="8" col_id="8" name="billed_charges" label="billed_charges" type="double"
      rawvalue="QHmkAAAAAAA=" index="IDX" just="r" colwidth="8" scale="0" precision="0" />
  </row>
- <row event_name="row" trigger_name="attr_out" output_name="Print" output_label="Data Set
    WORK.CLAIM_DETAIL" section="body" colcount="1" index="IDX" just="c">
```

Definition of the new tagset "claims" begins by issuing the PROC TEMPLATE command and specifying the name and storage location of the new tagset—in this case in the "tagsets" library. The INDENT attribute specifies the number of spaces that NDENT and XDENT statements within the PROC TEMPLATE step indent the associated tag.

```
PROC TEMPLATE;
DEFINE TAGSET Tagsets.claims /STORE=sasuser.templat;
INDENT=5;
```

Within the PROC TEMPLATE step, the DEFINE statements control the syntax that will be produced when the corresponding event is processed by ODS. For example, when the "claims" tagset encounters the "doc" event, the XML declaration "<?XML version="1.0" encoding="windows-1252"?> will be written to the output file along with a comment indicating that the output file is being produced for submission of claims data. The PUTQ statement writes out the current ENCODING option value and places it in quotes. The NL argument of a PUT statement specifies that the tagset should insert a new line in the output file.

```
DEFINE EVENT doc;
       START:
            /*put required declarations in your XML*/
             PUT  "<?xml version=""1.0""";
             PUTQ " encoding=" ENCODING;
             PUT  " ?>" NL;
             PUT  "<!--" ;
             PUT "  XML File for Submission of Claims Data";
             PUT "  -->" NL NL;
             PUT "<claims xmlns=""http://www.cdms-llc.com/ns/sasclaims/1.0"">" nl;
```

The EVAL statement "creates or updates a user-defined variable by setting the value of the variable to the return value of a WHERE expression" (SAS, 2011b, page 1177). In this case, a line counter variable is being created so that we can output a claim line-item number in our claims dataset. Within each claim_id, the tagset will restart counting the detail records associated with that claim and assign a line-item detail number to the output dataset[5]. The "START:" and "FINISH:" event statuses are used to control the writing of tags at the beginning and end of an event, respectively. In this example, the START of the "doc" event triggers the writing of the XML declaration and the "claims" root tag and the FINISH triggers writing the closing tags for the <claim> and <claims> tags.

```
             NDENT;
             EVAL $counter 0;
             EVAL $curr_claim "-";
      FINISH:
             XDENT;
             PUT "</claim>" NL;
             XDENT;
             PUT "</claims>" NL;
END;
```

Following definition of the "doc" event, the "row" and "data" events are defined. If the row event encountered contains the value "body" for the "section" attribute, processing continues; otherwise the event is skipped and processing of the PROC or DATA step output continues with the next event. Note that the IF/THEN logic in the TEMPLATE LANGUAGE is a bit different than in BASE SAS. The conditional test follows a "/" and in this example the IF logic is read as "IF the comparison of the "section" attribute's value to the literal string value "body" yields TRUE then…". In BASE SAS, this same syntax would be expressed as "IF section = "body" THEN…".

```
DEFINE EVENT row;
  START:

      TRIGGER data /IF CMP(section,"body");
END;
```

Within the "data" event (which is a child of the "row" event), the name attribute is assessed to determine if the data element being processed is the "claim_id". If it is, then the value of "curr_claim" is assessed to determine if the row

[5] A tagset is not an ideal way to achieve this numbering because it requires users of the tagset to know that the data being processed with the tagset should be sorted and processed in a certain order. The example is utilized here to show the power of tagsets to control processing—even generating new data elements based on the data found in the output of the DATA or PROC step.

being processed is either (a) the first row of data from the source document ($curr_claim = "-") or (b) the first claim for the current claimid (where the current value of the "value" attribute is not the same as $curr_claim). If either of these conditions is true, then the value of "line_num" is set to "1". If this "claim_id" element is not the first claim_id in the file, a closing tag "</claim>" is written to close the previous claim record and a new "<claim>" tag is opened along with "<claimid>" and "<detail>" tags.

```
DEFINE EVENT data;
     START:
     DO /IF CMP(name,"claim_id") ;
        DO /IF CMP($curr_claim,"-") or
              (^CMP($curr_claim,"-") and ^CMP($curr_claim,value));
           EVAL $line_num 1;
           XDENT ;
           PUT "</claim>" NL /IF ^CMP($curr_claim,"-");
           SET $curr_claim value ;
           NDENT ;
           PUT "<claim>" NL ;
           NDENT;
           PUT "<claimid>" VALUE "</claimid>" NL;
           NDENT;
           PUT "<detail>" NL ;
```

If the current row being processed is not the first detail record for a given "claim_id", then the line_num value is incremented by 1.

```
        ELSE ;
           DO /IF CMP($curr_claim,value) ;
              EVAL $line_num $line_num + 1;
               PUT "<detail>" nl ;
           DONE;
         DONE;
     DONE;
```

If the data element being processed is not "claim_id", the "name" attribute of the current data element is evaluated to determine the tag and associated value to write to the output file. Note that when the "account_no" data element is processed both the "<line>" and "<account_no>" elements are written to the file.

```
     PUT "<line>" $line_num "</line>" NL /IF cmp(name,"account_no") ;
     PUT "<acct_no>" VALUE /if cmp(name,"account_no");
     PUT "</acct_no>" NL /if cmp(name,"account_no");
     PUT "<member>" VALUE /if cmp(name,"member");
     PUT "</member>" NL  /if cmp(name,"member") ;
     PUT "<billed_charges>" VALUE /if cmp(name,"billed_charges") ;
     PUT "</billed_charges>" NL /if cmp(name,"billed_charges") ;
     PUT "</detail>" NL /IF CMP(name,"billed_charges");
  END;
END;
RUN;
```

With the new "claims" tagset built, our flat "claim_detail" dataset can now be output to our exact specifications.

```
PROC SORT DATA=work.claim_detail;
 BY claim_id member;
RUN;

ODS MARKUP TYPE=claims FILE='c:\_data\claim_detail_ods.xml';

PROC PRINT DATA=work.claim_detail;
RUN;
ODS MARKUP CLOSE;
```

```
<?xml version="1.0" encoding="WINDOWS-1252"?>
<!-- XML File for Submission of Claims Data -->
- <claims xmlns="http://www.cdms-llc.com/ns/sasclaims/1.0">
  - <claim>
      <claimid>584723988U</claimid>
    - <detail>
        <line>1</line>
        <acct_no>XWY3928957</acct_no>
        <member>Smith, Michael</member>
        <billed_charges>192.40</billed_charges>
      </detail>
    - <detail>
        <line>2</line>
        <acct_no>XWY3928957</acct_no>
        <member>Smith, Michael</member>
        <billed_charges> 25.20</billed_charges>
      </detail>
    - <detail>
        <line>3</line>
        <acct_no>XWY3928957</acct_no>
        <member>Smith, Michael</member>
        <billed_charges> 18.90</billed_charges>
      </detail>
  </claim>
  - <claim>
      <claimid>587439204T</claimid>
    - <detail>
        <line>1</line>
        <acct_no>VGH3344562</acct_no>
        <member>Jones, Mary</member>
```

This XML file can also be created through the XML LIBNAME Engine by specifying the tagset to be used in writing the dataset.

```
LIBNAME clmdtl XML 'C:\_data\ claim_detail_ods.xml' TAGSET=claims;

DATA clmdtl.claim_detail;
 SET work.claim_detail;
RUN;
```

The idea that tagsets would be used to drive the structure of the XML files output by the XML Libname Engine should come as no surprise when you consider the XMLTYPE option described earlier. One of the most obvious differences between the different XMLTYPEs (e.g., ORACLE, MSACCESS) is the tagging that is used to describe the elements. In fact, when both the XMLTYPE and TAGSET options are used to define an XML LIBNAME, the TAGSET takes precedence in determining the tagging and file structure. In the following example, the ORACLE XMLTYPE is specified in creation of the output file "claims", but because the "claims" tagset is also specified, the output is written to the tagset specifications instead of the XMLTYPE specifications.

```
LIBNAME CLAIMS XML '\\datasrvr\oracle vs tagset.xml' XMLTYPE=ORACLE TAGSET=claims;

DATA claims.claim_detail;
 SET work.claim_detail;
RUN;
```

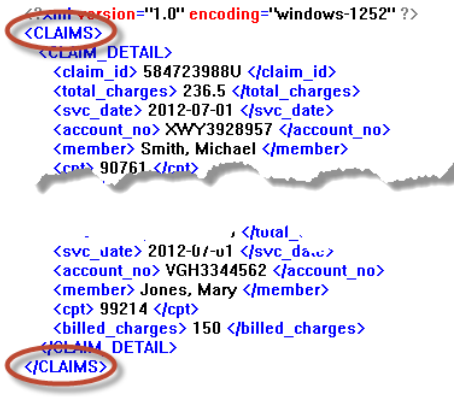
XMLTYPE=ORACLE	XMLTYPE=ORACLE & TAGSET=claims
<?xml version="1.0" encoding="windows-1252" ?> <ROWSET> <ROW> <claim_id> 584723988U </claim_id> <total_charges> 236.5 </total_charges> <svc_date> 2012-07-01 </svc_date> <account_no> XWY3928957 </account_no> <member> Smith, Michael </member> <cpt> 82310 </cpt> <billed_charges> 18.9 </billed_charges> </ROW> <ROW> <claim_id> 584723988U </claim_id> <total_charges> 236.5 </total_charges> <svc_date> 2012-07-01 </svc_date>	<?xml version="1.0" encoding="windows-1252" ?> <!– XML File for Submission of Claims Data –> <claims xmlns="http://www.cdms-llc.com/ns/sasclaims/1.0"> <claim> <claimid>584723988U</claimid> <detail> <acct_no>XWY3928957</acct_no> <line>1</line> <member>Smith, Michael</member> <billed_charges>192.4</billed_charges> </detail> <detail> <acct_no>XWY3928957</acct_no> <line>2</line> <member>Smith, Michael</member> <billed_charges>25.2</billed_charges> </detail>

The previous example might cause you to wonder about the default XML tags used by the XML Libname Engine and how they can be altered to achieve some of your XML-customization goals. By default the XML Libname Engine writes all root tags as "<TABLE>". However, by altering the tagset "SASXMOG" you can create a tagset with a custom root tag. In the following example, the "claimsroot" tagset is created based on the PARENT tagset "SASXMOG"; the existing "PUT <TABLE>" and "PUT </TABLE>" tags are simply replaced with the desired root tags "<CLAIMS>" and "</CLAIMS>", respectively.

```
PROC TEMPLATE;
   DEFINE TAGSET tagsets.claimsroot;
     PARENT=tagsets.sasxmog;
        DEFINE EVENT SASTable;
           START:
              PUT "<CLAIMS>";
              BREAK;
           FINISH:
              PUT "</CLAIMS>";
              BREAK;
        END;
   END;

RUN;
```

When using a PARENT tagset as the basis for you custom tagset, the event definitions specified in your new tagset take precedence over the definitions in the PARENT, but events not handled by your new tagset are handled by the PARENT—to the extent that the PARENT contains definitions for those events. As a result, the new custom tagset in this example (claimsroot) will produce output with the same characteristics as the original default tagset with the exception of the root tag.

```
LIBNAME CLAIMS XML 'C:\_data\claims root.xml'  tagset=claimsroot;

DATA claims.claim_detail;
 SET work.claim_detail;
RUN;
```



SAS (2010b) provides an even more flexible solution to this issue of assigning a user-defined root tag and provides an example of the ability of PROC TEMPLATE to utilize SAS Macro variables. In the following example—adapted from SAMPLE: 38052, the MVAR statement is used to bring the value of the macro variable "RootTagName" into the tagset "dynamic_root". Then, in the definition of the SASTable event, the value of RootTagName is conditionally assigned as the tag name based on whether the macro variable value is null or not null [/IF !EXIST(RootTagName)][6]. If RootTagName has a null value or is not found in the global symbol table, the default value of TEXT ("TABLE") in the default tagset is used as the root element.

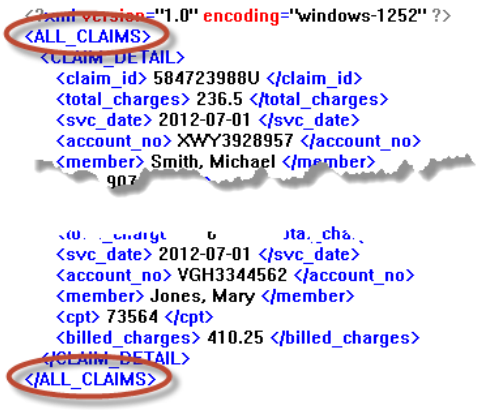
```
PROC TEMPLATE;
   DEFINE TAGSET tagsets.dynamic_root;
     PARENT=tagsets.sasxmog;
        MVAR RootTagName;
        DEFINE EVENT SASTable;
           START:
              PUT "<";
              DO / IF !EXIST(RootTagName);
             PUT TEXT;
              ELSE;
```

[6] Note also that within the tagset, the macro variable name is not prepended with "&" (Zender, 2006).

```
            PUT RootTagName;
          DONE;
              PUT ">" NL;
              BREAK;
           FINISH:
              PUT "</";
              DO / if !exist(RootTagName);
            PUT TEXT;
          ELSE;
            PUT RootTagName;
          DONE;
              PUT ">" NL;
              BREAK;
        END
   END;
RUN;
```

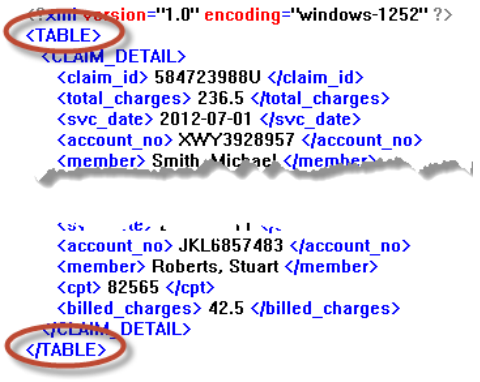
With this new tagset compiled, the following code will generate XML with the root tag "<ALL_CLAIMS>".

```
%LET RootTagName=ALL_CLAIMS;

LIBNAME CLAIMS XML 'C:\_data\new default xml tagset.xml'  TAGSET=dynamic_root;

DATA claims.claim_detail;
 SET work.claim_detail;

RUN;
```



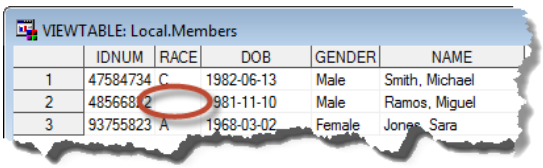
With a NULL value assigned to the macro variable, the root tag name returns to the default "TABLE".

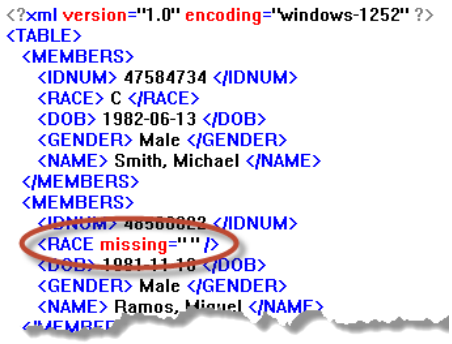
```
%LET RootTagName=;

LIBNAME CLAIMS XML 'C:\_data\new default xml tagset.xml'  TAGSET=dynamic_root;

DATA claims.claim_detail;
 SET work.claim_detail;
RUN;
```



Another event processed by the default XML tagsets is the occurrence of missing values. In the following example, you can see that writing a missing value through the LIBNAME Engine, by default, produces an element with a "missing=" attribute.

```
LIBNAME members XML 'c:\_data\membersB.xml';
DATA members.members;
 SET local.members;
RUN;
```





By altering the "SASXMOG" definition of the "SASColumn" and "MLEVDAT" events, SAS (2004, Usage Note: 23621) shows how tags containing missing values can be removed altogether. In the following definition of the SASColumn event, if the value for the current column is missing, an interrupt (BREAK) is executed ❶ and processing continues to the MELVDAT trigger ❷. Here, if the value is null/missing another BREAK is issued ❸, and procsessing passes back to the "FINISH" section of the SASColumn event—where again, the variable is assessed to determine if the value is missing ❹. In this new tagset, in the event of a missing value, no PUT statements are executed and no tag is written in associating with the missing value. Conversely, if a variable's value is not null, an opening tag is written by PUTting "<", followed by the variable's name ❺, putting a closing bracket ">" after the name, and writing the variable's value ❻ and the closing tag ❼.

```
PROC TEMPLATE;
DEFINE TAGSET tagsets.missing_tags;
       PARENT = tagsets.sasxmog;
❷ DEFINE EVENT MLEVDAT;
          ❸ BREAK     / IF EXISTS(MISSING);
          ❻ PUT       '>'         ;
            PUT       VALUE       ;
            BREAK;
  END;
  DEFINE EVENT SASColumn;
       START:
          ❶ BREAK     / IF EXISTS(MISSING);
            NDENT;
            PUT       '<' ;
          ❺ PUT       NAME;
            BREAK;
      FINISH:
          ❹ BREAK     / IF EXISTS(MISSING);
            PUT       '</';
          ❼ PUT       NAME;
            PUT       '>'  CR;
            XDENT;
            BREAK;
  END;
END;
RUN;
```

Note in the description of the previous tagset that execution of the events does not follow the physical order in which they are written in PROC TEMPLATE, but rather the order in which the events occur during execution of the PROC or DATA step that is utilizing the tagset. After compiling the new tagset "missing_tags" it is used in the following library definition to produce an XML file that does not write a tag for missing values.

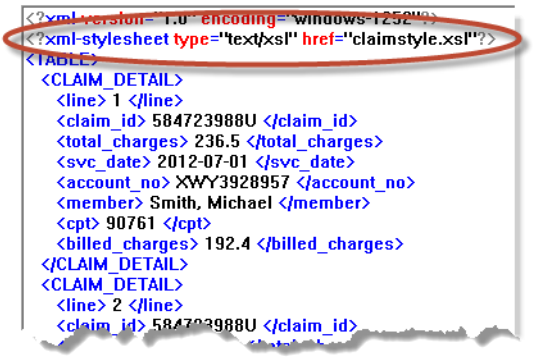
```
LIBNAME members XML 'c:\_data\members.xml' TAGSET=missing_tags;

DATA members.members;
 SET local.members;
RUN;
```

```
<?xml version="1.0" encoding="windows-1252" ?>
<TABLE>
  <MEMBERS>
    <IDNUM>47584734</IDNUM>
    <RACE>C</RACE>
    <DOB>1982-06-13</DOB>
    <GENDER>Male</GENDER>
    <NAME>Smith, Michael</NAME>
  </MEMBERS>
  <MEMBERS>
    <IDNUM>48566822</IDNUM>
    <DOB>1981-11-10</DOB>
    <GENDER>Male</GENDER>
    <NAME>Ramos, Miguel</NAME>
  </MEMBERS>
  <MEMBERS>
    <IDNUM>93755823</IDNUM>
    <RACE>A</RACE>
    <DOB>1968-03-02</DOB>
    <GENDER>Female</GENDER>
    <NAME>Jones, Sara</NAME>
  </MEMBERS>
</TABLE>
```

Another tagset alteration you might want to apply to the default tagsets (or as a modification to your own user-defined tagsets) is to add a reference to an Extensible Style Language Transformation (XSLT) stylesheet. As has been discussed throughout the paper, XML is used mainly as a medium of data interchange—not as a means of formatting data for presentation. XSLT, on the other hand, is used to transform XML files. These transformations can be used to create yet another XML file (Castro & Goldberg, 2009), or, as in the following example can be used to transform the XML file into an HTML file (Morgan, 2011). In this application of XSLT, the data elements are repackaged (transformed) into another tagging structure to allow the data to be rendered for display in a web browser. Not surprisingly, the XSLT file used to perform this transformation is itself, an XML file. The root element "xsl:stylesheet" identifies the document as a set of XLST instructions to be applied to an XML file. Next, the "xsl:template" specifies the tag to which the XSLT language is applied—in this case, the "TABLE" tag. After specification of these XSLT elements, the "HTML" and "BODY" tags familiar to HTML developers appear along with the syntax specifying an HTML table. After the headings for the table are specified, the values that will populate the table are referenced relative to the tags in the associated XML file that contain them. The XSLT syntax will loop through each "claim_detail" parent element and select the child element values to populate the HTML table.

```
<?xml version="1.0" encoding="windows-1252"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0">
<xsl:template match="/TABLE">
<html>
<body>
<table border="1">
    <tr> <th>ClaimID</th>
         <th>Line</th>
         <th>Svc. Date</th>
         <th>Account #</th>
         <th>Member</th>
         <th>CPT</th>
         <th>Billed Charges</th>
    </tr>
    <xsl:for-each select="CLAIM_DETAIL">
    <tr>
         <td><xsl:value-of select="claim_id"/></td>
         <td><xsl:value-of select="line"/></td>
         <td><xsl:value-of select="svc_date"/></td>
         <td><xsl:value-of select="account_no"/></td>
         <td><xsl:value-of select="member"/></td>
         <td><xsl:value-of select="cpt"/></td>
         <td><xsl:value-of select="billed_charges"/></td>
    </tr>
</xsl:for-each>
</table>
</body>
</html>
</xsl:template>
</xsl:stylesheet>
```

In order for this stylesheet to be applied to an XML file, the XML file must specify the XSLT file in a processing instruction. In the following example, the "claim_detail" XML file has been altered to include the xml-stylesheet instruction that references the preceding stylesheet "claimstyle.xsl".



With both the .XSL and .XML written to the web directory, navigating to the .XML file results in the following HTML page being rendered in the web browser.

ClaimID	Line	Svc. Date	Account #	Member	CPT	Billed Charges
584723988U	1	2012-07-01	XWY3928957	Smith, Michael	90761	192.4
584723988U	2	2012-07-01	XWY3928957	Smith, Michael	82565	25.2
584723988U	3	2012-07-01	XWY3928957	Smith, Michael	82310	18.9
587439204T	1	2012-07-01	VGH3344562	Jones, Mary	73564	410.25
587439204T	2	2012-07-01	VGH3344562	Jones, Mary	99214	150
866978852B	1	2012-07-11	JKL6857483	Roberts, Stuart	73564	383.12
866978852B	2	2012-07-11	JKL6857483	Roberts, Stuart	73564	192
866978852B	3	2012-07-11	JKL6857483	Roberts, Stuart	82310	18.9
866978852B	4	2012-07-11	JKL6857483	Roberts, Stuart	82565	42.5

However, in order to reproduce this result with future versions of the "claim_detail" XML file, the processing instruction needs to be written to the XML file as it is generated from SAS. In order to do this, all you need to do is add the processing instruction to the DOC event of a tagset as follows.

```
PROC TEMPLATE;
  DEFINE TAGSET tagsets.apply_xsl;
    PARENT=tagsets.sasxmog;
    DEFINE EVENT doc;
    START:
      PUT "<?xml version=""1.0""";
      PUTQ " encoding=" encoding;
      PUT "?>" NL NL;
      PUT '<?xml-stylesheet type="text/xsl" href="claimstyle.xsl"?>' ;
    END;
  END;
RUN;
```

Then, using the tagset, simply write the XML file with the XSLT processing instruction.

```
LIBNAME CLAIMS XML 'C:\XML\apply xsl stylesheet.xml'  tagset=apply_xsl;

DATA claims.claim_detail;
 SET work.claim_detail;
RUN;
```

These tagset examples have merely scratched the surface with respect to what is possible with the ODS MARKUP destination and PROC TEMPLATE. With PROC TEMPLATE, it is possible to write SAS datasets out to almost any XML specification you encounter. For an excellent, in-depth discussion of writing XML templates for the XML libname engine the reader is referred to Zender (2006). In addition one should also consider the "custom" tagset specification provided by SAS (2010a). This tagset provides a great starting point from which to begin your exploration of writing tagsets for XML.

CONCLUSION

XML as a medium of data interchange is quickly changing the way many of us operate in the data management, analytic, and reporting arenas. As SAS users, we need to understand the basics of XML, develop expertise with SAS methods for both reading and writing these (sometimes complex) data sources. Hopefully this paper has helped frame the discussion in a manner that facilitates your continued learning about this very important topic and assists you in developing The Power to Know™.

REFERENCES

Castro, E. & Goldberg, K.H. (2009). XML: Visual QuickStart Guide. Berkeley, CA: Peach Pit Press.

Cox, T.W. (2012). Advanced XML Processing with SAS® 9.3. Proceedings of the SAS Global Forum 2012. Cary, NC: SAS Institute, Inc.

Haworth, L.E., Zender, C.L., & Burlew, M.M. (2009). Output Delivery System: The Basics and Beyond.

Martell, C. (2008). SAS® XML Mapper to the Rescue. Proceedings of the SAS Global Forum 2008. Cary, NC: SAS Institute, Inc.

Morgan, J. (2011). Simple XSLT Tutorial: XSLT in 5 Minutes. http://www.youtube.com/watch?v=BujLy71JY1k.

SAS Institute Inc. (2004). Usage Note 23621: With the XML engine, how can I only output a node if it's not missing?. http://support.sas.com/kb/23/621.html

SAS Institute Inc. (2010a). SAS® 9.2 XML LIBNAME Engine: User's Guide, Second Edition. Cary, NC: SAS Institute, Inc.

SAS Institute Inc. (2010b). Sample 38052: How to modify the root node generated by default when creating output using the XML LIBNAME engine. http://support.sas.com/kb/38/052.html. Cary, NC: SAS Institute, Inc..

SAS Institute Inc. (2011a). SAS® 9.3 XML LIBNAME Engine: User's Guide. Cary, NC: SAS Institute, Inc..

SAS Institute Inc. (2011b). SAS® 9.3 Output Delivery System: User's Guide. Cary, NC: SAS Institute, Inc.

SAS Institute Inc. (2012). SAS® 9.3 XML LIBNAME Engine: User's Guide, Second Edition. Cary, NC: SAS Institute, Inc..

Schacherer, C. (2012). The FILENAME Statement: Interacting with the World Outside of SAS®. Proceedings of the SAS Global Forum 2012. Cary, NC: SAS Institute, Inc.

Schacherer, C. (2013). The SAS® Programmer's Guide to XML and Web Services. Proceedings of the SAS Global Forum 2013. Cary, NC: SAS Institute, Inc.

Shoemaker, J.N. (2005). XML Primer for SAS® Programmers. . Proceedings of the 30th Annual SAS Users Group International Meeting. Cary, NC: SAS Institute, Inc.

World Wide Web Consortium (2006). XML Markup Language (XML) 1.1, Second Edition. http://www.w3.org/TR/2006/REC-xml11-20060816/.

World Wide Web Consortium (2008). XML Markup Language (XML) 1.0, Fifth Edition. http://www.w3.org/TR/2008/REC-xml-20081126/.

Zender, C. (2006). Creating a Tagset Template for the SAS® XML Libname Engine. Proceedings of the Pharmaceutical Industry SAS® Users Group 2006.

CONTACT INFORMATION

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

Christopher W. Schacherer, Ph.D.
Clinical Data Management Systems, LLC
Madison, WI 53711
Phone: 608.478.0029
E-mail: CSchacherer@cdms-llc.com
Web: www.cdms-llc.com