

## The SAS® Programmer's Guide to XML and Web Services

Christopher W. Schacherer, Clinical Data Management Systems, LLC

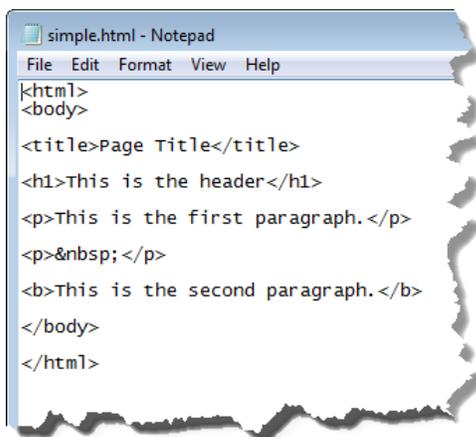
### ABSTRACT

Because of XML's growing role in data interchange, it is increasingly important for SAS programmers to become familiar with SAS technologies and techniques for creating XML output, importing data from XML files, and interacting with web services—which commonly use XML file structures for transmission of data requests and responses. The current work provides detailed examples of techniques you can use to integrate these data into your SAS solutions using XML Mapper®, the XML LIBNAME engine, Output Delivery System®, the FILENAME statement, and new SOAP functions available beginning in SAS 9.3.

### 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 markup 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.

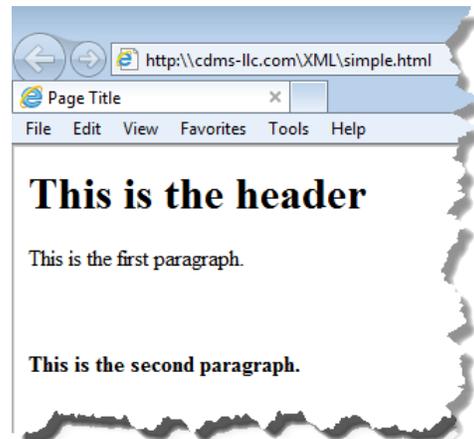


```

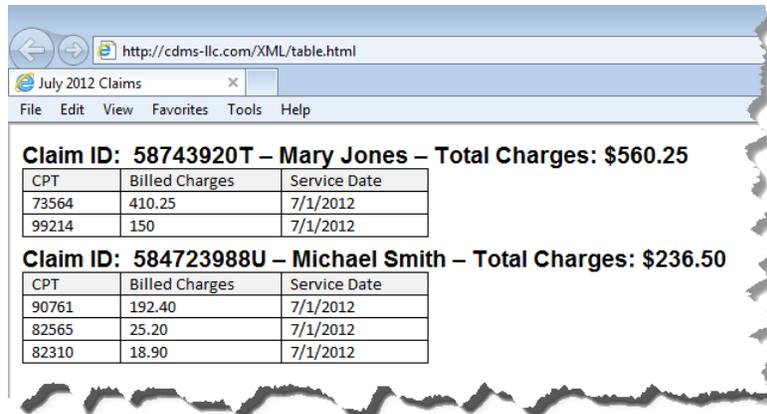
simple.html - Notepad
File Edit Format View Help
<html>
<body>

<title>Page Title</title>
<h1>This is the header</h1>
<p>This is the first paragraph.</p>
<p>&nbsp;</p>
<b>This is the second paragraph.</b>
</body>
</html>

```



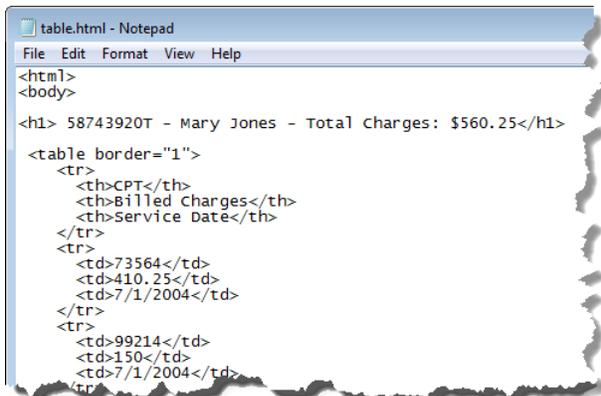
In an example more closely resembling a real-world application of using HTML to markup and present a dataset to an end-user, consider the following list of medical claims in which the claim ID, member name, and total charges serve as a header record followed by an HTML table of the line-item details 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 the 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.



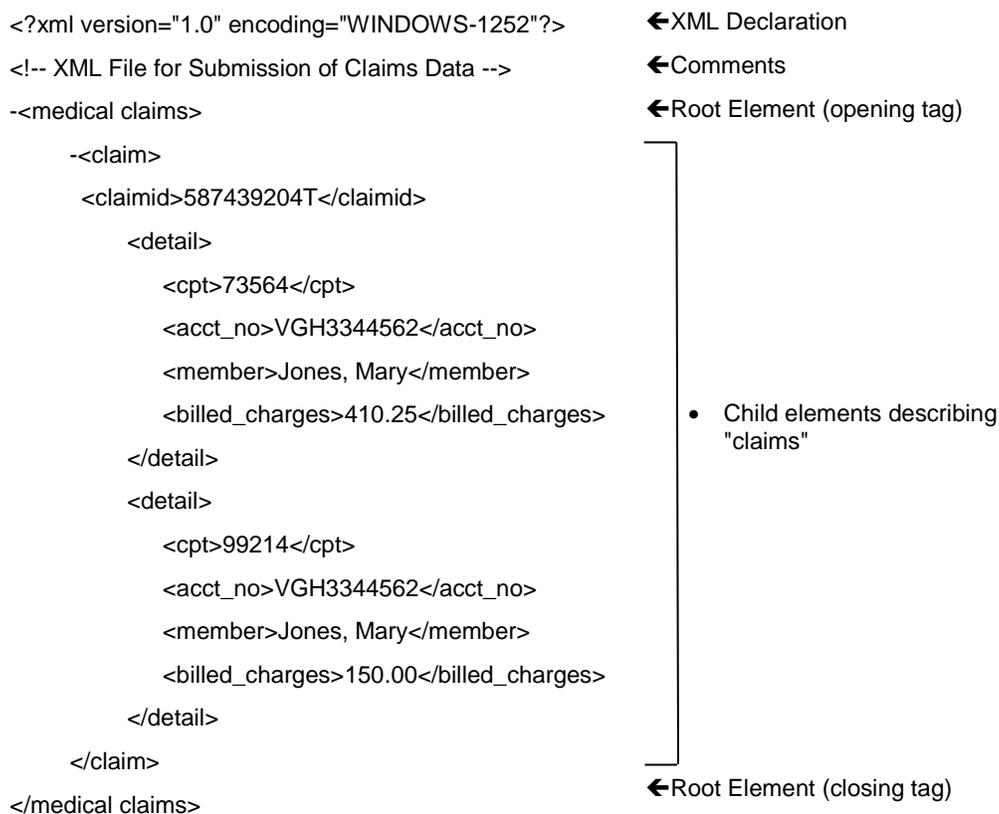
	member_name	claimid	total_charg	html_text
1	Mary Jones	58743920T	\$560.25	<td>73564</td>
2	Mary Jones	58743920T	\$560.25	<td>410.25</td>
3	Mary Jones	58743920T	\$560.25	<td>7/1/2004</td>
4	Mary Jones	58743920T	\$560.25	<td>99214</td>
5	Mary Jones	58743920T	\$560.25	<td>150</td>
6	Mary Jones	58743920T	\$560.25	<td>7/1/2004</td>

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.



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<sup>1</sup>. 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

<sup>1</sup> 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<sup>2</sup>.

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

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,

---

<sup>2</sup> 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 of the data and the second-level element (or, node) as the "dataset".

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 -->
<claims xmlns="http://www.cdms-llc.com/ns/sasclaims/1.0"> ❷
  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.<sup>3</sup> The following two XML files result in identical SAS datasets.

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

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.

<sup>3</sup> 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 we can create an infinite number of unique markup "languages" 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 these techniques, however, you should first understand the parameters that bound the default functionality of the XML LIBNAME engine.

First, the preceding XML LIBNAME examples all used the default XMLTYPE "GENERIC". Because this XMLTYPE is the default, the following declarations are identical in their functionality.

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

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

In addition to GENERIC, there are other XMLTYPE values that facilitate data integration by complying with specific, proprietary XML standards when writing XML data and by utilizing information from those standards when reading XML files that use them (e.g., ORACLE, MSACCESS, CDISC). However, the differences in these XMLTYPEs have mainly to do with tag structures and utilization of metadata and less with overcoming the challenges provided by complex tag structures. In addition to the XMLTYPE, there are also XML LIBNAME options that provide additional support for managing the import/export of numeric data (XMLDOUBLE), allow the dataset's encoding to be overridden when writing an XML file (XMLENCODING), and even some that enable data management tasks that can run afoul of the XML standard but which may simplify production data management tasks (XMLCONCATENATE<sup>4</sup> & XMLPROCESS).

Despite these available options, however, it can still be challenging to import and export data to and from XML files when the structure of the XML file overwhelms the XML engine's ability to reliably and accurately translate the XML tag structure into a SAS dataset. With this issue in mind, you should note that the GENERIC XMLTYPE is capable of reliably reading 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.

<sup>4</sup> Concatenate actually allows multiple XML declarations to exist in the same file so that you can append multiple files with the same XML structure (or, for that matter, different XML structures) into a single file and read it through the XML engine. Note that this is NOT well-formed XML per the W3C standard, but it definitely has practical value in performing day-to-day production data management tasks.

```

<?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>
</claims>

```

```
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:  
c:\\_data\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, 2010; 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<sup>5</sup>. 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 'c:\_data\claim_detail.xml' XMLMAP='c:\claim translator.map';

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

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 XML maps is the tag "SXLEMAP" ❷. This root element contains all of the instructional parameters that tell the

<sup>5</sup> 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.

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" ③. 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 element "/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 ④. Similarly, the XPATH locations of each column are specified within each "COLUMN" element ⑤, and the name of each column is assigned via the "name" attribute ⑥. Data types and lengths of the variables can also be assigned within each COLUMN tag ⑦.

```

    ② <?xml version="1.0" encoding="WINDOWS-1252"?> ①
    - <SXLEMAP version="1.2">
      - <TABLE name="claim"> ③
        <TABLE-DESCRIPTION>claim</TABLE-DESCRIPTION>
        <TABLE-PATH syntax="XPath">/claims/claim</TABLE-PATH> ④
        - <COLUMN name="claimid"> ⑥
          <PATH syntax="XPath">/claims/claim/claimid</PATH> ⑤
          <TYPE>character</TYPE>
          <DATATYPE>string</DATATYPE> ⑦
          <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 with the specification of the "detail" table and ends with the closing tag of the "SXLEMAP" element.

```

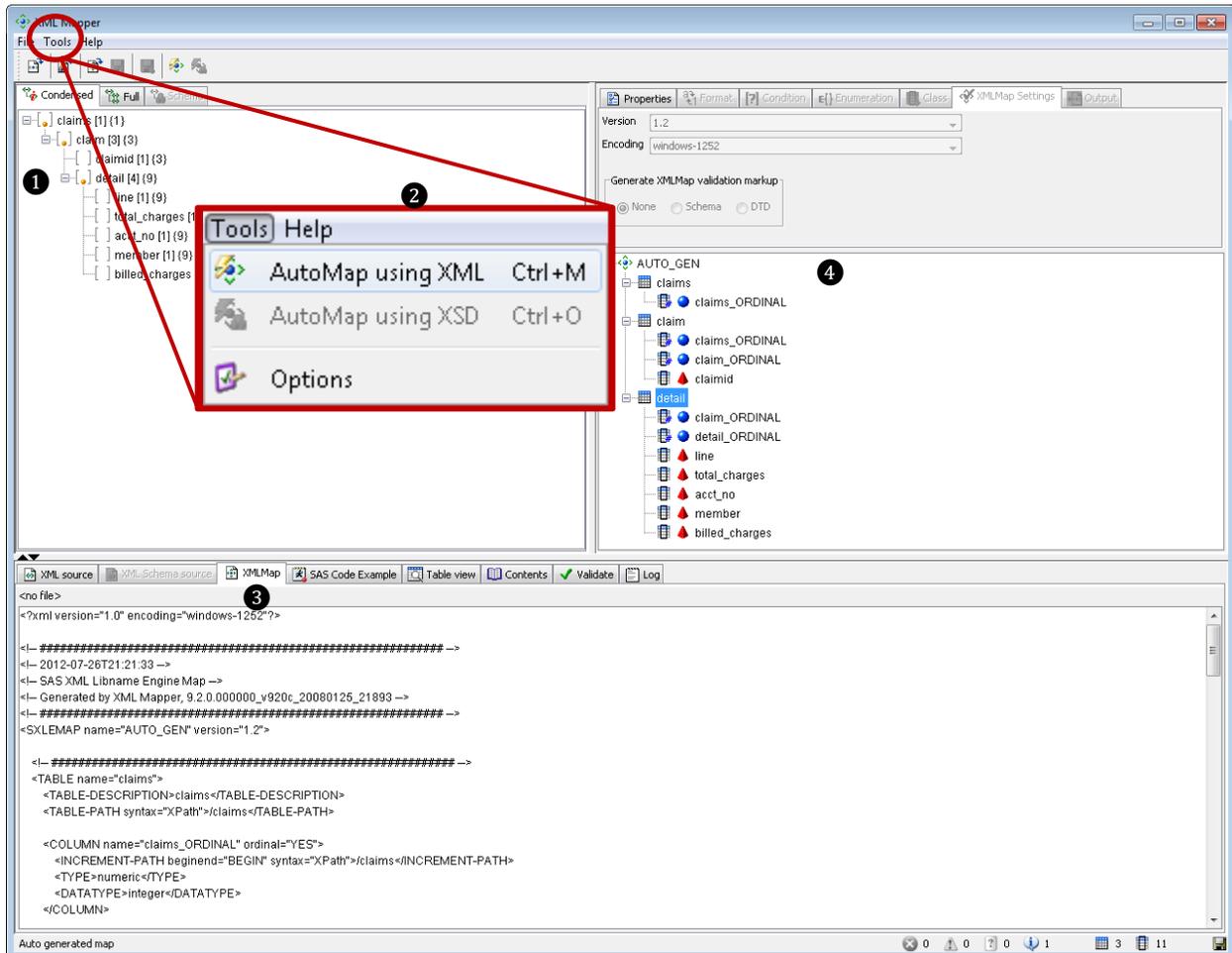
    ...
    - <TABLE name="detail">
      <TABLE-DESCRIPTION>detail</TABLE-DESCRIPTION>
      <TABLE-PATH syntax="XPath">/claims/claim/detail</TABLE-PATH>
      - <COLUMN name="claimid" retain="YES">
        <PATH syntax="XPath">/claims/claim/claimid</PATH>
        <TYPE>character</TYPE>
        <DATATYPE>string</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 very user-friendly way of creating these .map files.

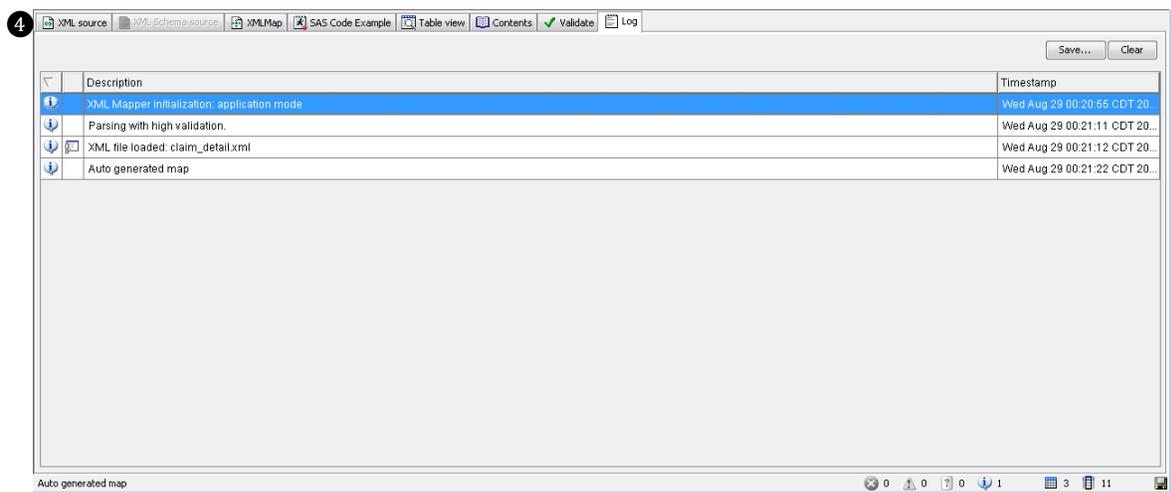
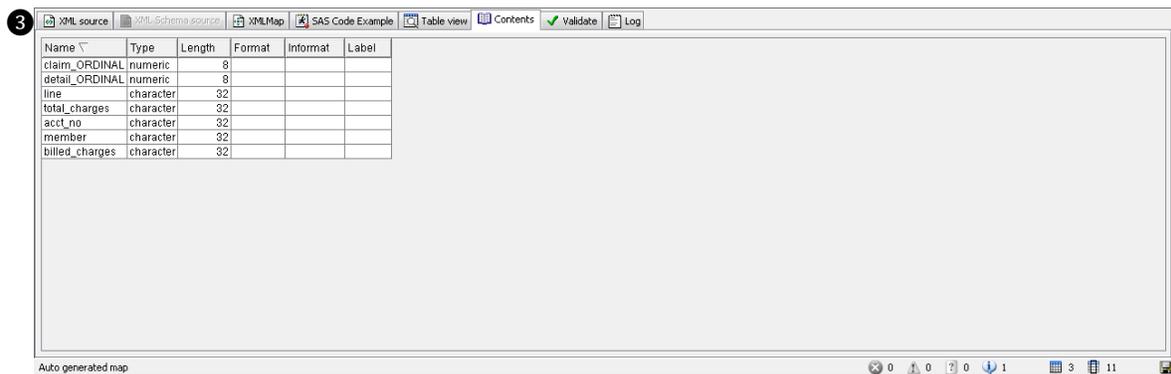
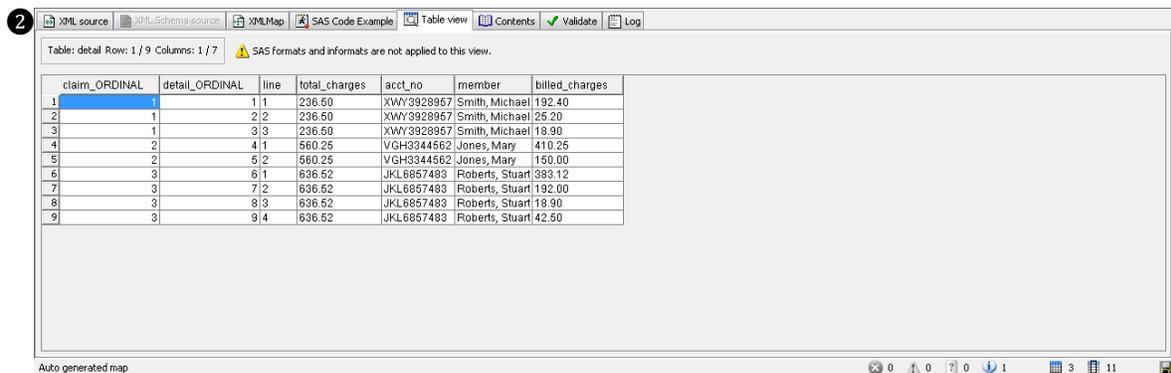
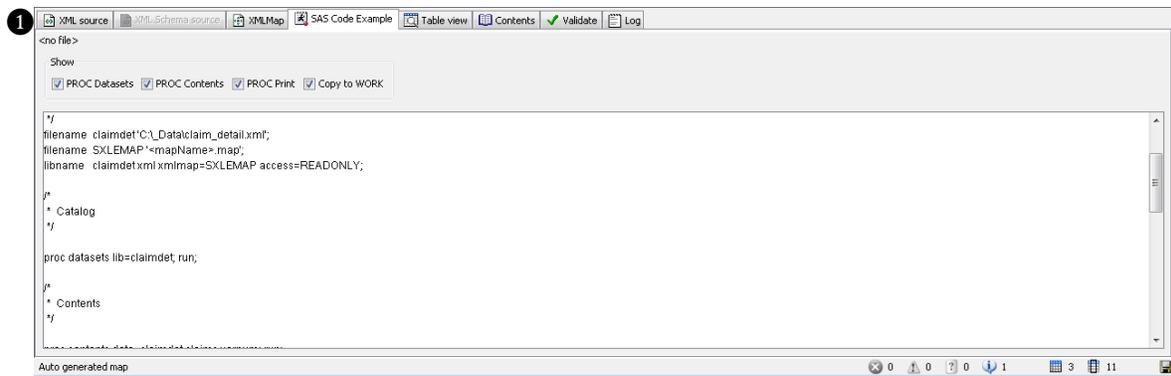
## XML MAPPER

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

Upon first opening an XML source file, you notice that the levels of the hierarchy within the file are represented in the source file pane in the upper left of the interface ❶. In response to selecting the "AutoMap" function in the tools menu ❷, XML Mapper attempts to create an XML Map file ❸. In the tabbed XML Map Properties pane in the upper right of the interface, a representation of the data table structure created by the XML Map is presented ❹.

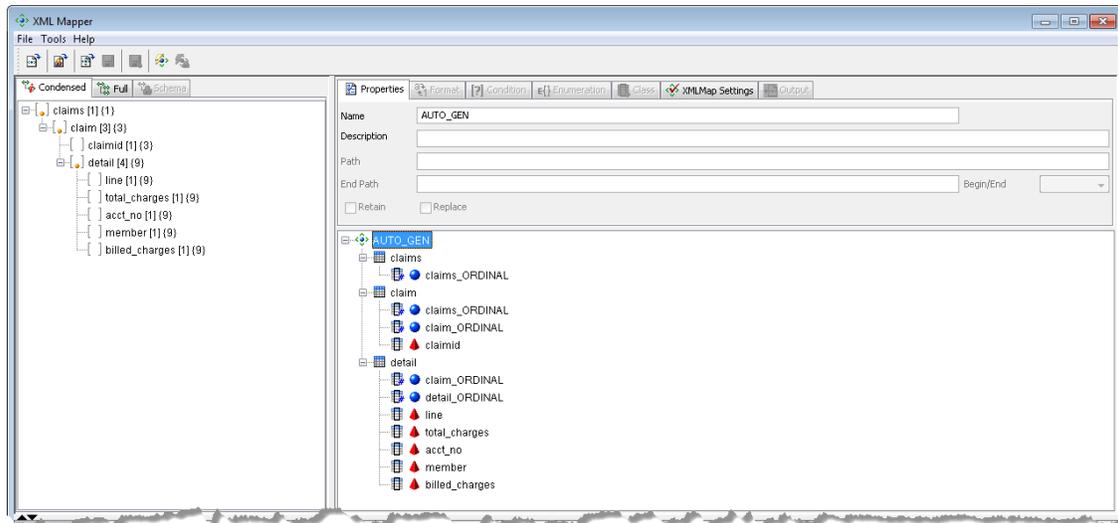


Starting from this AutoMapped instruction set (i.e., the XML Map), you can see all of the other functionality provided by XML Mapper in helping you create the mapping file. In the following views of the result pane at the bottom of the interface, we see, in addition to the XML Map, example code for interacting with the datasets resulting from accessing the XML file through a library defined with the XML LIBNAME engine using the current XML Map ❶, a view of the actual data in the dataset currently selected in the AutoMapping detail pane ❷, the names, data types, and lengths of the variables in that dataset ❸, and a log of the activity that has transpired during your XML Mapper session ❹.

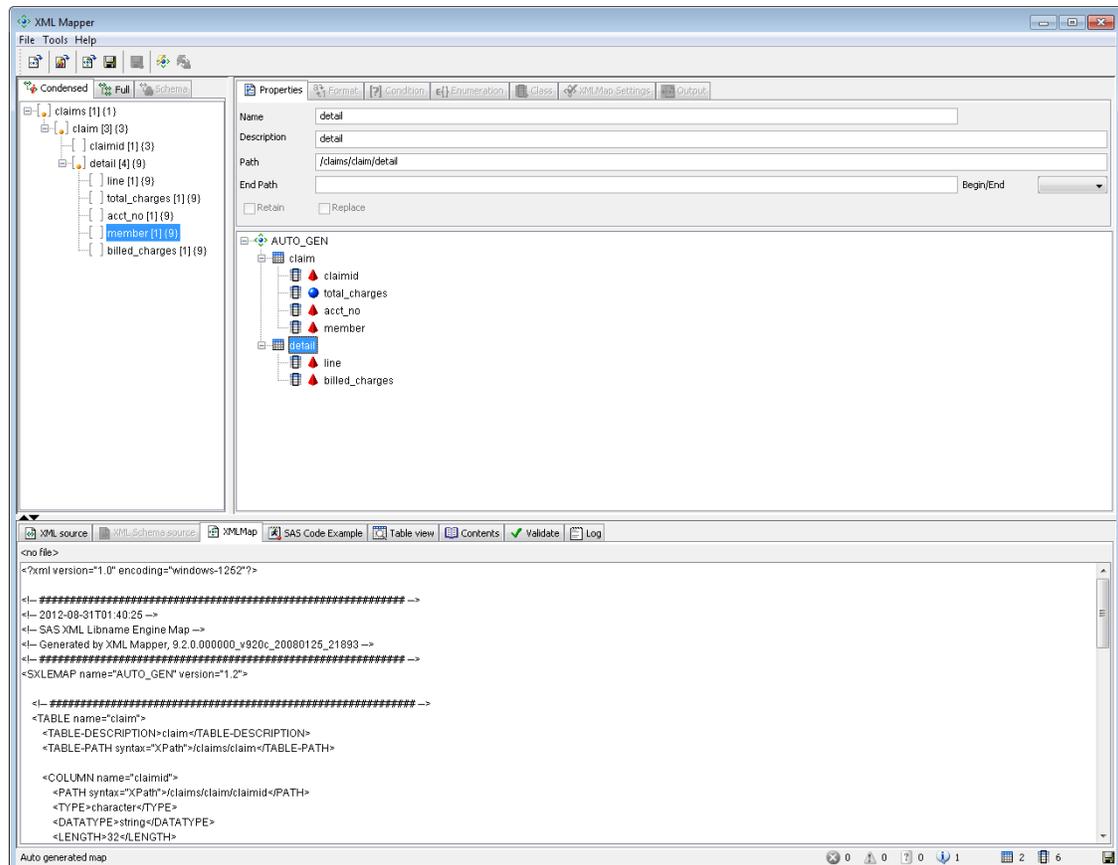


In addition to automatically creating a default mapping file, however, the XML Mapper is also interactive in that you can add or remove elements to or from the tables represented in the table structure of the default map and the

changes to the mapping file are rewritten on the fly. In the following example, the default map for our healthcare claims data shows that under the default mapping several characteristics of the overall claim (e.g., account\_no, total\_charges, and member) will be rendered in the "detail" dataset.



Knowing our data well, we are confident that these elements can safely be moved to the "claim" dataset without harming the integrity of the data, so we simply delete them from the "details" table structure on the right and drag-and-drop the XML elements from the source file pane on the left to the table representation in the XML Map Properties pane. When we do so, the XML map is automatically rewritten to reflect our new designation of the fields that should be represented in the "claim" and "detail" datasets.



Notice also, that because the "claims" dataset that results from the AutoMap function is simply the single record associated with the root tag, it was deleted from our updated map file. Similarly, as the "line" variable serves to uniquely identify each claim line-item record within a given claim, we also removed "detail\_ordinal" from the "detail" dataset. 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 primary/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 the ability to manually declare the variables included in 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.

The screenshot shows the XML Mapper interface with the 'Format' tab selected. The 'Format' field is set to '\$' and the 'Width' is 12 and 'NDec' is 2. A preview table titled 'VIEWTABLE: Local.Claims' shows the resulting data with the 'total\_charges' column formatted as currency.

	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 and control the variable values through Enumeration (which controls the assignment of allowed and missing values) and Conditional processing (which can be used to conditionally value variables based on element attributes).

The screenshot shows the XML Mapper interface with the 'Description' field set to 'Billed Charges Total'. A preview table titled 'VIEWTABLE: Local.Claims' shows the resulting data with the 'Billed Charges Total' column labeled as such.

	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

When you have reviewed the resulting SAS datasets and are comfortable that your mapping file is correct and adequately fulfills your needs, simply save the XML Map (File → Save XMLMap As) and specify its location in your XML LIBNAME definition. Once defined, this LIBNAME can be used to read data from your XML file.

```
LIBNAME clmfile XML 'c:\_data\claim_detail.xml' XMLMAP='c:\claim translator.MAP';
DATA local.claims;
SET clmfile.claim;
RUN;
```

Now that you can access data from a wide variety of XML files with shapes both simple and complex, how can you use this new skill? Of course, it is useful to be able to produce and read XML files if you are producing XML files to send (via FTP, E-Mail, etc.) to vendors or regulatory agencies. Likewise, if colleagues and business partners are using batched XML files to deliver canned datasets these new skills will come in handy for parsing and loading the file into an analytic dataset. However, one of the most useful applications of these skills is the ability to interact with web services—which typically use XML as a method of data interchange.

## ACCESSING DATA VIA WEB SERVICES

A web service is "a software system designed to support interoperable machine-to-machine interaction over a network." (W3C, 2004). The purpose of this arrangement, of course, is so that owners of one system can provide some functionality, some information to another system in a manner that allows both systems to remain separate and autonomous. The interchange of data is made through agreement to make requests of the source data system in a specific manner as prescribed by the source system owner and accept responses in a prescribed fashion that adheres to a set of rules that can be used to parse the data. Typically, this interaction involves the use of XML to either make the request, return the result set, or both.<sup>6</sup>

### REST ARCHITECTURES

There are two main architectural approaches for enabling web services—REST and SOAP<sup>7</sup>. REST stands for Representational State Transfer. As described by Chinthaka (2007), when accessing a REST-based (or, RESTful) web service you are specifying a particular view (or representational state) of the available resource. As you change the parameters that specify which representation of a resource you wish to receive (through clicking on a link, selecting a value from an interface component, or specifying a particular URL) your representational state is changed (or, transferred).

REST is based on a simple set of HTTP commands (get, post, put, and delete) and therefore, REST requests are usually rendered as a URL that identifies the representation you wish to access from the specified resource. In the following example, a request for information about National Drug Code (NDC) value "0067-2000-91" is sent to the "getcode" web service at hippaspace.com. Because the return type (rt) is specified as XML, the file returned in response to this request is an XML file.<sup>8</sup> If you save that file to a local disk or network directory, you can easily access it via an XML library using the previously-discussed methods. However, in accessing a web service in the context of data analysis using SAS, this approach seems (in most cases) to defeat the purpose of using a web service. What you probably want to do is call the web service in order to integrate data available from the web service with your SAS dataset(s). Therefore, you need to be able to call the web service from within SAS so that the entire process of requesting the data and converting it to a SAS dataset can all be accomplished from the same SAS program.

In this example (adapted from Mack, 2010 and demonstrated previously by Schacherer, 2012), the request to the "getcode" web service is made via the URL access method of the SAS FILENAME statement and the XML file returned from the web service is converted into a SAS dataset via the XML LIBNAME engine. In order to send a request to the "getcode" service using the URL access method, the parameters expected by the web service are assigned to the macro variables "ndc", "return\_type", and "token"—representing, respectively, the NDC code for which information is being requested, the form of the data to be returned in response to the request, and a security token that identifies you as an authorized user<sup>9</sup>. In the following macro variable assignments, %QSYSFUNC executes the URLENCODE function in a manner that masks special characters, and the URLENCODE function encodes special characters that might otherwise impact resolution of a URL. The %NRSTR and %SUPERQ functions are then used to assign the value of macro variable "target". %NRSTR is used to mask the special characters in components of the URL that will represent the parameter inputs and %SUPERQ puts the values of the parameters in quotes.

---

<sup>6</sup> Through use of these technologies, we begin to work toward Richardson and Ruby's (2007) vision of the the web as a "network that you can use whether you're serving data to human beings or computer programs." The early days of the web saw a rapid expansion of technologies and methods geared toward making data more accessible to human consumers of information. Pretty soon, the web that presented data to humans was somewhat at odds with the "programmable web"—a term Richardson & Ruby use to describe "programmer-friendly technologies for exposing a web site's functionality in officially sanctioned ways—RSS, XML-RPC, and SOAP."

<sup>7</sup> It should be noted that these methodologies are not necessarily orthogonal to one another and that some REST web services are technically also reliant on SOAP solutions.

<sup>8</sup> It should be noted that these parameters (i.e., rt, token, q) are specific to the "getcode" web service at hippaspace.com and are not generalizable parameters for searching web services—although "q=" is somewhat standard as a query parameter. The provider of the web service in which you are interested in using should have documentation available on which parameters to specify and how to call the particular web service.

<sup>9</sup> As of the time the paper was written, the security token in this example was available from hipaaspace.com to allow prospective users to test their web services.

```

%LET ndc          =%QSYFUNC (URLENCODE (0067-2000-91));
%LET return_type =%QSYFUNC (URLENCODE (xml));
%LET token       =%QSYFUNC (URLENCODE (3932f3b0-cfab-11dc-95ff-0800200c9a663932f3b0-
                                cfab-11dc-95ff-0800200c9a66));

%LET target=%NRSTR (http://www.HIPAASpace.com/api/ndc/getcode) %NRSTR (?q=) %SUPERQ&ndc
                %NRSTR (&rt=) %SUPERQ (return_type) %NRSTR (&token=) %SUPERQ (token);

```

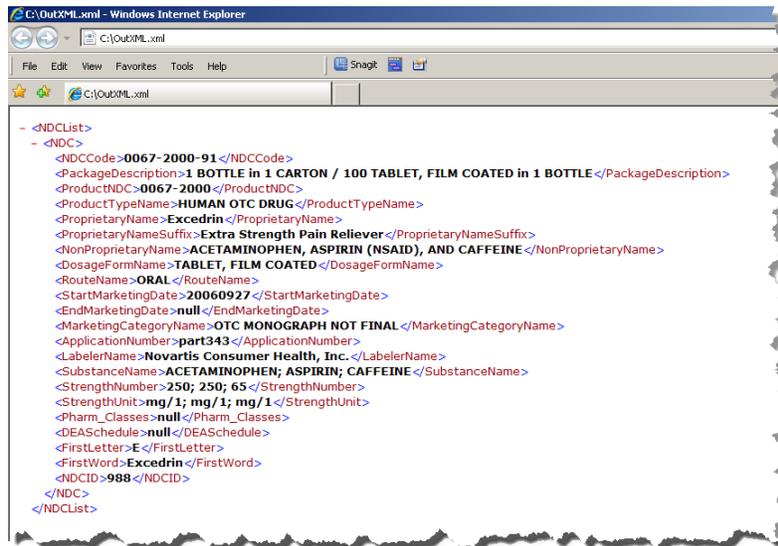
Following assignment of the &target macro variable, the file reference "inurl" is defined as the URL corresponding to the value of &target. The XML file "c:\outxml.xml" is referenced by the file reference "outxml", to which the data returned by the web service will be written. The subsequent DATA\_NULL\_ step connects to the specified URL—making the web service request via the INFILE statement—and, after reading in the returned data lines with the INPUT statement, writes (or, FILEs) the lines of XML syntax returned from the web service to the file referenced by "outxml".

```

FILENAME inurl URL "%SUPERQ(target)" LRECL=4000 DEBUG;
FILENAME outxml "c:\OutXML.xml";

DATA_NULL_ ;
INFILE inurl LENGTH=len;
INPUT record $varying4000. len;
FILE outxml NOPRINT NOTITLES RECFM=n;
PUT record $varying4000. len;
RUN;

```

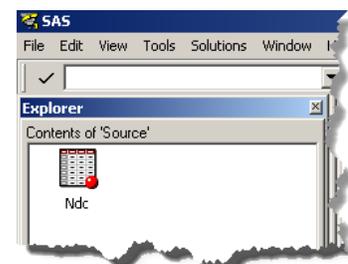


With the resulting XML file saved locally, the SAS library "source" is created with the XML LIBNAME engine pointing to the local file containing the response to the web service request.

```
LIBNAME source XML 'C:\outxml.xml' ACCESS=READONLY;
```

Note that, in the SAS Explorer you can see the NDC dataset in the "source" library, but the SAS XML engine does not support double-clicking on the XML "datasets" to open them in browse mode—as is noted in the following error—even though the data are available for analysis and reporting via DATA and PROC steps<sup>10</sup>.

ERROR: The PROC has requested that the XML Engine perform an unsupported function. As a work-around you might consider copying the data into the WORK library and having the PROC process from the copy.



<sup>10</sup> Some exceptions to this statement, not mentioned elsewhere in the paper, are that PROC SQL UPDATE statements and PROC SORT are not allowed to operate against these data—though both of these PROCs can be used on a persistent local copy of the data once they become a SAS dataset.

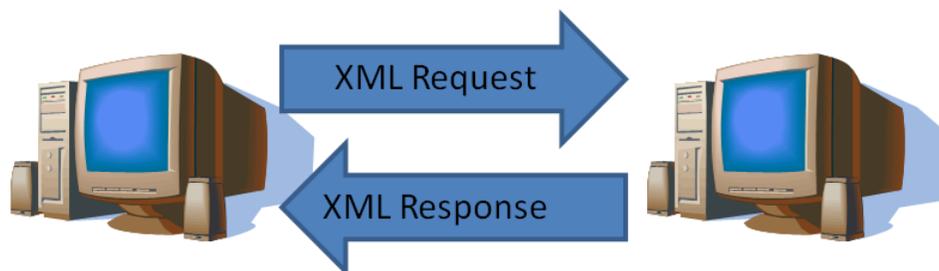
Of course you can, as in the previous examples, save a copy of the dataset to another physical library—creating a persistent SAS dataset that can be browsed in the SAS Explorer.

```
DATA work.ndc_from_xml;
  SET source.ndc;
RUN;
```

	NDCID	FIRSTWORD	FIRSTLETTER	DEASCHEDULE	PHARM_CLASSES	STRENGTHUNIT	STRENGTHNUMBER	SUBSTANCENAME	LABELERNAME
1	988	Excedrin	E	null	null	mg/1; mg/1; mg/1	250; 250; 65	ACETAMINOPHEN; ASPIRIN; CAFFEINE	Novartis Consumer H

## SOAP-BASED WEB SERVICES

Unlike RESTful web services (in which you are indicating the "representation" of the data you wish to use), when using a SOAP-based web service, you send a request in the form of a message, the web service interprets that message and generates another message in reply. Both of these messages are typically XML, so you need to be able to not only read XML through the XML LIBNAME engine, but also produce an XML message to send as your request. The form of the request is specified by the web service provider and can be derived from the Web Service Definition Language (WSDL) file that describes the web service(s) available from the specific provider. A discussion of the WSDL language is beyond the scope of this paper, but see Mack (2010) for an excellent description of using third party tools (e.g., SOAP UI) to create syntactically correct SOAP requests based on the WSDL file.



In the following example, the XML request that will be sent is based on the XML "shell" depicted below. This XML syntax is referred to as a shell because it contains all of the static text that will be part of the SOAP request, but it also contains a reference to a SAS macro variable that will be resolved just prior to sending the request to the web service—allowing the SAS program calling the web service to dynamically populate the request with different values of the macro variable. Specifically, this XML shell will be used to access the "QueryItem" service at hipaaspace.com and it specifies that the request being made is for an NDC code. The NDC code for which we are requesting information will be specified by resolution of the macro variable "&NDCCode" and the security/access token is hard-coded to the same value used in the previous REST example.

```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:ns="http://HIPAASpace.com/webservice/2008">
  <soapenv:Header/>
  <soapenv:Body>
    <ns:QueryItem>
      <!--Optional:-->
      <ns:Type>NDC</ns:Type>
      <!--Optional:-->
      <ns:SearchRequest>&NDCCode</ns:SearchRequest>
      <!--Optional:-->
      <ns:Token>3932f3b0-cfab-11dc-95ff-0800200c9a663932f3b0-cfab-11dc-
95ff-0800200c9a66</ns:Token>
    </ns:QueryItem>
  </soapenv:Body>
</soapenv:Envelope>
```

This request syntax is saved as the file "C:\NDC Request.txt" on the local system running SAS. In the SAS program making the SOAP request, the value of the macro variable "NDCCode" is assigned as "055045-1807-\*9"—the NDC code that will ultimately be sent in the SOAP request, and the fileref "request" references the shell of the XML request. Next, the filerefs "tempreq" and "response" are defined with the TEMP access method. The TEMP access method "creates a temporary file that exists only as long as the filename is assigned. The temporary file can be accessed only through the logical name and is available only while the logical name exists" (SAS, 2011b). The

TEMP access method is used here to avoid retaining persistent files for the requests and responses being sent to/from the web service. Each time the following DATA step is run, a new version of the temporary request file "tempreq" will be generated with the current value of "&NDCCode", and the corresponding response from the QueryItem service will be written to a new, temporary "response" file.

```
%LET NDCCode=055045-1807-*9;

FILENAME request 'C:\NDC Request.txt';
FILENAME tempreq TEMP;
FILENAME response TEMP;
```

Specifically, following assignment of the filerefs, a DATA \_NULL\_ step identifies fileref "tempreq" as a target file to which data will be written and the fileref "request" as a source dataset from which data will be read. The INPUT statement then fetches each line from the "request" file (i.e., the XML shell), but instead of reading the lines of XML syntax into a SAS dataset (this is a DATA \_NULL\_ step), the INPUT statement simply fetches each line of text into the automatic variable "\_INFILE\_". The value of \_INFILE\_ (which contains the entire line of data read from the "request" file) is then assigned to the local variable "local\_infile" which, in turn, is written to the temporary file "tempreq" using the PUT statement.

```
DATA _NULL_ ;
  FILE tempreq;
  INFILE request;
  INPUT;

  local_infile = RESOLVE(_INFILE_);

  PUT local_infile;
RUN;
```

So, for this example (with &NDCCode having a value of "055045-1807-\*9"), the temporary file tempreq has the following contents:

```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:ns="http://HIPAASpace.com/webService/2008">
  <soapenv:Header/>
  <soapenv:Body>
    <ns:QueryItem>
      <!--Optional:-->
      <ns:Type>NDC</ns:Type>
      <!--Optional:-->
      <ns:SearchRequest>055045-1807-*9</ns:SearchRequest>
      <!--Optional:-->
      <ns:Token>3932f3b0-cfab-11dc-95ff-0800200c9a663932f3b0-cfab-11dc-
95ff-0800200c9a66</ns:Token>
    </ns:QueryItem>
  </soapenv:Body>
</soapenv:Envelope>
```

Next, PROC SOAP is used to send the contents of the "tempreq" temporary file to the web service identified by the location of the web service definition (URL) and the name of the requested service (SOAPACTION).

```
PROC SOAP IN=tempreq
  OUT=response
  URL="http://www.hipaaspace.com/wspHVLookup.asmx"
  SOAPACTION='http://HIPAASpace.com/webService/2008/QueryItem';
RUN;
```

Unlike the previous REST example, however, the response from this web service does not meet the rectangular shape requirement necessary to define an XML library without the aid of an XML map. Fortunately, XML Mapper can be used to help construct an XML map based on the response file. However, you will have to make a slight variation to the preceding code in order to generate a named physical file containing the XML response.

Simply change the file reference for "response" in the previous example from the TEMP access method to the DISK access method and rerun the DATA \_NULL\_ step and the PROC SOAP step.

```
FILENAME response 'C:\Response.xml';
```

Following this change, the response file will be written to a file that you can open with XML Mapper and from which you can generate a MAP file. Once generated, save the .map file as "C:\NDCMap.map" and use it to define a SAS

library using the XML LIBNAME engine.

```
LIBNAME response XML XMLMAP='C:\NDCMap.map'
```

As in the previous REST example, you can then operate on these data in much the same way as you would datasets in any other SAS LIBRARY.

```
DATA work.NDC_Code_Data;  
  SET response.pair;  
RUN;
```

## SOAP FUNCTIONS

In the previous example a PROC SOAP step was used to specify the XML request and response file names, identify the URL and SOAP action (i.e., web service), and make the SOAP call. Beginning in SAS 9.3, however, there is a way to execute SOAP requests within the DATA step itself—via the new SOAP functions. In the following example, an analytic program begins by making a request to the web service "getlookups" in order to download the lookup tables from a medical record system for later use in creation of user-defined formats using PROC FORMAT. After defining the file reference for the request—in this case, a static SOAP request to receive all of the current production lookup table value-pairs—a DATA \_NULL\_ step is executed in which the URL and the web service (SOAPACTION) are assigned as the value of local variables. These local variables, in turn, are used as inputs to the SOAPWEB function along with the file references specifying the SOAP request and response files. After receiving the response from this web service request, an XML LIBNAME can be assigned to the response file and the value/label pairs can be used as the input to a PROC FORMAT step.

```
FILENAME request 'C:\lookup table request.txt';  
FILENAME response TEMP;  
  
DATA _NULL_ ;  
  
  URL="http://www.hospitalxyz.org/emrLookup.asmx";  
  SOAPACTION='http://www.hospitalxyz.org/ws/bi-ops/GetLookUps';  
  rc=SOAPWEB("request",url,"response",soapaction,,,,,);  
  
RUN;
```

In addition to the four inputs shown in this example (IN, OUT, URL, and SOAPACTION), SOAPWEB accepts additional inputs for username, password, and domain (for basic web authentication) and host, port, username, and password (for HTTP proxy server authentication) (SAS, 2011c). In addition, the input "MUSTUNDERSTAND" allows a 0 or 1 to be passed in the SOAP message to specify whether the SOAP processor must process the header entries (1) or not (0). The CONFIGFILE and DEBUG input parameters, respectively, are used to (a) "specify a character value that is a SPRING configuration file...used primarily to set time-out values" and (b) "specify a character value that is the full path to a file that is used for debugging logging output." For the SOAPWEB function, the only required input parameters are the "IN" (request) and "URL". The other SOAP functions (SOAPWEBMETA, SOAPWIPSERVICE, etc.) are all variants of the basic SOAP functionality—i.e., to make SOAP calls and direct responses—and vary mainly in terms of their approach to communicating authentication and meta-data/configuration information between the SAS client and the web service.

In addition to using static SOAP requests, however, the introduction of the SOAP functions might help you start thinking about SOAP calls in a new, more data-driven way. Consider the following form of requests to the "finclass" web service. Replacing the empty "patient identifier" tag (<pid> ? </pid> ) with a macro variable reference would allow you to make SOAP calls in a manner similar to the previous NDC example that utilized a PROC SOAP call to interact with the QueryItem web service.

```
<soapenv:Envelope  
  xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"  
  xmlns:tem="http://tempuri.org/"  
  xmlns:msg="http://www.cdms-llc.com/DM/2012/financials/messages">  
  <soapenv:Header/>  
  <soapenv:Body>  
    <tem:finclass>
```

```

<tem:request>
  <msg:pid?></msg:pid>
  <msg:UserName?></msg:UserName>
  <msg:Password?></msg:Password>
</tem:request>
</tem:finclass>
</soapenv:Body>
</soapenv:Envelope>

```

However, "finclass" (like many web services) allows one or more instances of an element ("pid" in this case) to be submitted in a single request. So, if you were creating an analytic dataset and were missing the "financial class" for (say) three patients, you could send a SOAP request similar to the following to get the financial class for all three missing data points:

```

<soapenv:Envelope
  xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:tem="http://tempuri.org/"
  xmlns:msg="http://www.cdms-llc.com/DM/2012/financials/messages">
  <soapenv:Header/>
  <soapenv:Body>
    <tem:finclass>
      <tem:request>
        <msg:pid>586321</msg:pid>
        <msg:pid>869570</msg:pid>
        <msg:pid>213515</msg:pid>
        <msg:UserName>cschacherer</msg:UserName>
        <msg:Password>mysecretpassword</msg:Password>
      </tem:request>
    </tem:finclass>
  </soapenv:Body>
</soapenv:Envelope>

```

The following response file is received, and it conforms to the XML LIBNAME engine's rule of providing a rectangular dataset, so you can make a local SAS dataset copy of the data using the SAS syntax, below.

```

<?xml version="1.0"?>
<finclass xmlns="http://tempuri.org/">
  <finclassresult>
    <pid>586321</pid>
    <class>A</class>
  </finclassresult>
  <finclassresult>
    <pid>869570</pid>
    <class>C</class>
  </finclassresult>
  <finclassresult>
    <pid>213515</pid>
    <class>S</class>
  </finclassresult>
</finclass>

```

```

LIBNAME classdat XML 'c:\finclass_response.xml' ;
DATA work.finclass;
  SET classdat.finclassresult;
RUN;

```

But how was the request file with multiple "pid" elements generated? In the previous SOAP examples, there was a single request being made and the specific request sent was generated by resolving a single macro variable in a single tag in the request shell. Now we need to dynamically create multiple instances of an element (with different values) in the middle of the request file. Whereas there are undoubtedly a number of ways that this could be achieved, the example provided below demonstrates a relatively simple way to generate the request file with a DATA \_NULL\_ step. First, a FILENAME statement is used to create references for both the SOAP request and the SOAP response. Then, a DATA \_NULL\_ step is executed to write the request file syntax to the "request" fileref<sup>11</sup>. When the request file syntax reaches the point where you are ready to write the tags identifying the patients for which you are requesting financial class data, a DO LOOP is opened, the dataset containing those patient IDs is read, and, based on the condition that the record has a missing data point for "financial\_class", a <pid> tag is written to the request file. After all of the necessary <pid> tags have been written, the remaining tags necessary to complete the request are written and the request file is ready to be sent to the web service for processing.

```

FILENAME request 'c:\request.xml' ;
FILENAME response 'c:\response.xml';

DATA _NULL_;
FILE request;
IF _N_ = 1 THEN DO;

    PUT '<soapenv:Envelope ' ;
    PUT '    xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/" ' ;
    PUT '    xmlns:tem="http://tempuri.org/" ' ;
    PUT '    xmlns:msg="http://www.cdms-llc.com/DM/2012/financials/messages">' ;
    PUT '    <soapenv:Header/>' ;
    PUT '    <soapenv:Body>' ;
    PUT '        <tem:finclass>' ;
    PUT '            <tem:request>' ;
END;

DO UNTIL (EOF_CLAIMS);
    SET claims END=EOF_CLAIMS;
    IF financial_class = '' THEN DO;
        XML_TAG = '        <msg:pid>' || pid || '</msg:pid>';
        PUT XML_TAG;
    END;
END;

    PUT '            <msg:useridd>cschacherer</msg:useridd>' ;
    PUT '            <msg:pwd>mysecretpassword</msg:pwd>' ;
    PUT '        </tem:request>' ;
    PUT '    </tem:finclass>' ;
    PUT ' </soapenv:Body>' ;
    PUT '</soapenv:Envelope>' ;

%GETFIN;

RUN;

```

However, because the request file "request.xml" is still open by the DATA \_NULL\_ step in which it is being written, the SOAPWEB function cannot use the request file reference "request" as the input parameter "IN". Instead of issuing the SOAPWEB function within this DATA \_NULL\_ step, the following MACRO is written to perform the SOAPWEB function call in the DATA \_NULL\_ step.

---

<sup>11</sup> NOTE: To determine the form of a valid request to the web service in which you are interested, you can use a tool like soapUI to interrogate the web service and find the proper form for your request or contact the administrator of the web service for the WSDL file—which can be used to determine the form of a valid request.

```

%MACRO GETFIN;
DATA _NULL_;
    URL="https://webservices.cdms-llc.com/financials.asmx";
    SOAPACTION='http://tempuri.org/finclass';
    rc=SOAPWEB("request",url,"response",soapaction,,,,,);
RUN;
%MEND GETFIN;

```

In addition to providing another example of using the SOAP functions within the DATA step, the preceding example introduces another important topic for those who want to become proficient in using SAS to interact with XML and web services—the ability to produce XML files with SAS.

## USING SAS TO WRITE XML FILES

Most of the examples so far have focused on reading XML files through the XML LIBNAME engine or interacting with web services. In the latter case, the sole example of writing an XML file used PUT statements to write the syntax of the XML file forming the web service request, and that method can be used very successfully to create even very intricate XML files. However, there are two additional methods for creating XML files that provide a more robust, reusable solution for the operational interchange of XML files—the XML LIBNAME engine and the SAS Output Delivery System (ODS).

### XML LIBNAME ENGINE OUTPUT

As demonstrated in one simple example previously, just as you can read data from a library based on the XML LIBNAME engine, you can also write data to an XML library in a manner that results in well-formed XML. In the following example, data from the healthcare claims dataset is written out to an XML file for transport to a third party data aggregator. 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>
  <- CLAIM_DETAIL
    <claimid> 587439204T </claimid>
    <total_charges> 560.25 </total_charges>
    <line> 2 </line>
  </TABLE>

```

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".

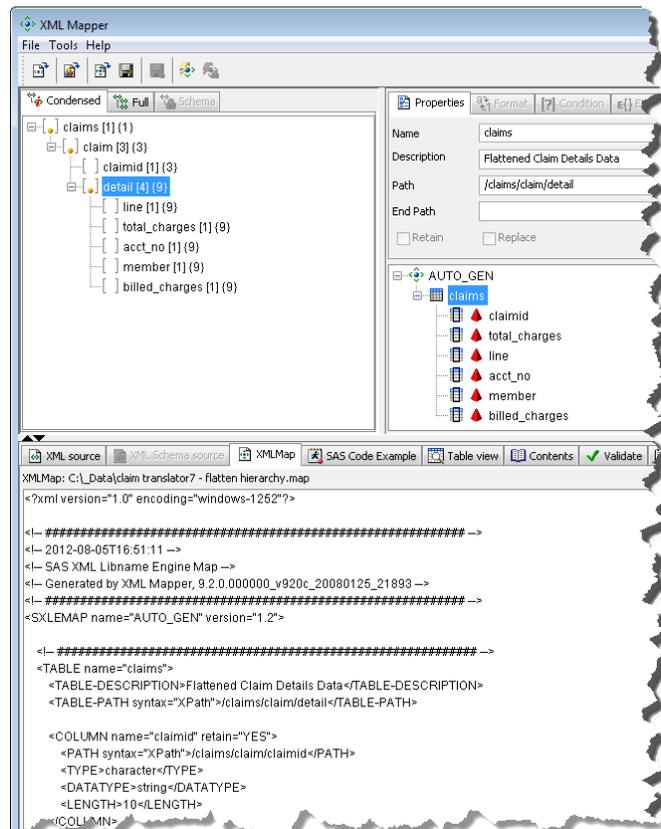
But what if the SAS dataset was originally created from the XML source file depicted below (with a hierarchical structure) and you want to write the data in the same hierarchical structure as the original source file? Perhaps you read the data in to make some programmatic changes to specific data values and now you need to output it in its original form—that is, the same XML structure.

```

<?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>

```

Assuming that you read these data in using an XMLMap, you can use the same mapping file (with some slight alterations) to reproduce the XML file that gave rise to your analytic dataset. As depicted here, XML Mapper was used to create an XMLMap that would flatten the source XML file to produce the two-dimensional claims dataset used in the preceding example. Specifically, the detail elements were dragged from the XML source elements list in the left pane to the XMLMap specification in the right pane. Then the "claimid" element was dragged to this same data table specification—resulting in the generation of the XMLMap syntax necessary to deliver the "claimid" and all of the detail elements in a single SAS dataset. Using the resulting XMLMap, as an input to an XML LIBNAME ENGINE definition, the hierarchical "claims" XML file was read into the SAS dataset "claims".



```

LIBNAME claimsin XML 'c:\claimdtl.xml'
      XMLMAP='c:\flat_claims.MAP';

```

```

DATA claims;
  SET claimsin.claims;
RUN;

```

With a few alterations, we can use this same XMLMap to output the data from the flattened dataset "claims" in a way that adheres to the hierarchical structure of the original file "claim\_detail.xml". With a text editor, an "OUTPUT" element specifying the data table to be output is added to the XMLMap "flat\_claims.map". In this case, the "claims" table is being output. Subsequently, when the claims data are written out to an XML file via the XML engine, the map will allow SAS to organize the output in the appropriate hierarchical levels according to the instructions originally used to read the data into the SAS dataset "claims".

```

claim translator7 - flatten hierarchy - out.map - Notepad
File Edit Format View Help
<?xml version="1.0" encoding="windows-1252"?>
<!-- ##### -->
<!-- 2012-08-05T16:44:16 -->
<!-- SAS XML Libname Engine Map -->
<!-- Generated by XML Mapper, 9.2.0.000000.v920c_20080125_21893 -->
<!-- ##### -->
<!-- ## Validation report ## -->
<!-- ##### -->
<!-- Column (claimid) in table (claims) has an XPath outside the scope
of the table path. The contents of this column may not correspond to
other row values and/or may be missing entirely. -->
<!-- XMLMap validation completed successfully. -->
<!-- ##### -->
<SXLEMAP name="flatten_claims" version="2.1">

  <OUTPUT>

    <TABLEREF name="claims" />

  </OUTPUT>

  <!-- ##### -->
  <TABLE name="claims">
    <TABLE-DESCRIPTION>Flattened Claim Details Data</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>
    </COLUMN>
  </TABLE>

```

In order to use this output method, you must specify the XMLV2 ENGINE nickname that utilizes enhancements to the XML LIBNAME engine that have been introduced subsequent to version 9.1.3. In addition to the ability to use an XMLMap for exporting from a SAS dataset to an XML file, the XMLV2 also provides support for XML namespaces and enforces XML compliance. (SAS, 2011a)

```

LIBNAME claimout XMLV2 'c:\_data\claim_out.xml' XMLMAP='c:\flat_claims-out.map';
DATA claimout.claims;
  SET work.claims;
RUN;

```

With the "claimout" library defined with the updated XMLMAP, the DATA step successfully writes the XML file "claim\_out.xml" with the same hierarchical structure as the original source file.

```

<?xml version="1.0" encoding="WINDOWS-1252"?>
<!-- SAS XML Libname Engine (SAS92XML) SAS XMLMap Generated
Output Version 9.03.01M0P06072011 Created 2012-08-05T21:19:34 -->
- <claims>
- <claim>
- <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>
  <claimid>584723988U</claimid>
</claim>
- <claim>
- <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>
  <claimid>584723988U</claimid>
</claim>
- <claim>
- <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>
  <claimid>584723988U</claimid>
</claim>
- <claim>
- <detail>
  <total_charges>560.25</total_charges>

```

With the combination of the XMLMap and XML LIBNAME engine, you can perform a number of very useful operations involving XML files. This combination of tools may very well meet all of your needs with respect to integrating XML into your daily operations. However, the Output Delivery System (ODS) can be utilized in cases where additional flexibility is required in generating XML files.

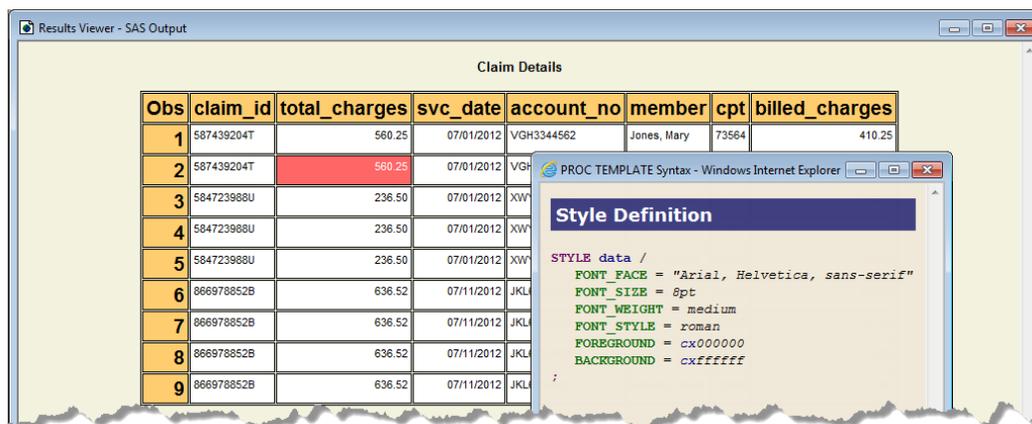
### ODS MARKUP TAGSETS

The Output Delivery System (ODS) "enables you to produce SAS procedure and DATA step output to many different destinations." (SAS, 2011d, p. 33). Common uses of ODS are to output 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 the 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 determines 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 fact/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;
```



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 - Notepad
File Edit Format View Help
<?xml version="1.0" encoding="windows-1252"?>
<?mso-application progid="Excel.sheet"?>
<Workbook xmlns="urn:schemas-microsoft-com:office:spreadsheet"
  xmlns:x="urn:schemas-microsoft-com:office:excel"
  xmlns:ss="urn:schemas-microsoft-com:office:spreadsheet"
  xmlns:html="http://www.w3.org/TR/REC-html40">
  <DocumentProperties xmlns="urn:schemas-microsoft-com:office:">
    <Author>Chris</Author>
    <LastAuthor>Chris</LastAuthor>
    <Created>2012-08-06T02:34:16</Created>
    <LastSaved>2012-08-06T02:34:16</LastSaved>
    <Company>SAS Institute Inc. http://www.sas.com</Company>
    <Version>9.03.01M0P06072011</Version>
  </DocumentProperties>
  <Styles>
    <Style ss:ID="_body">
      <Interior ss:Pattern="Solid" />
      <Protection ss:Protected="1" />
    </Style>
    <Style ss:ID="_contents">
      <Interior ss:Pattern="Solid" />
      <Protection ss:Protected="1" />
    </Style>
    <Style ss:ID="_pages">
      <Interior ss:Pattern="Solid" />
    </Style>
  </Styles>
</Workbook>

```

	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;

```

```

claim_details_csv.csv - Notepad
File Edit Format View Help
"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","VGH3344562","Jones, Mary","99214,150.00"
"3","584723988U","236.50,07/01/2012","XWV3928957","Smith, Michael","90761,192.40"
"4","584723988U","236.50,07/01/2012","XWV3928957","Smith, Michael","82565,25.20"
"5","584723988U","236.50,07/01/2012","XWV3928957","Smith, Michael","82310,18.90"
"6","866978852B","636.52,07/11/2012","JKL6857483","Roberts, Stuart","73564,383.12"
"7","866978852B","636.52,07/11/2012","JKL6857483","Roberts, Stuart","73564,192.00"
"8","866978852B","636.52,07/11/2012","JKL6857483","Roberts, Stuart","82310,18.90"
"9","866978852B","636.52,07/11/2012","JKL6857483","Roberts, stuart","82565,42.50"

```

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 (and even their content) 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="I">
- <doc_head event_name="doc_head" trigger_name="attr_out" class="body" just="I">
  <doc_meta event_name="doc_meta" trigger_name="attr_out" class="body" just="I" />
  <auth_oper event_name="auth_oper" trigger_name="attr_out" class="body" just="I" />
  <doc_title event_name="doc_title" trigger_name="attr_out" class="body" just="I" />
  <stylesheet_link event_name="stylesheet_link" trigger_name="attr_out" just="C" />
- <javascript event_name="javascript" trigger_name="attr_out" class="body" just="I">
  <startup_function event_name="startup_function" trigger_name="attr_out" class="startupfunction" just="I" />
  <shutdown_function event_name="shutdown_function" trigger_name="attr_out" class="shutdownfunction" just="I" />
</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 operator" event contains several data points that the "EVENT\_MAP" tagset instructs ODS to write out as attributes of the "doc operator" tag. However, in the previous EXCELXP example the tagset instruction with respect to the "doc operator" 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 instruction set provided to ODS in the form of the two tagsets.

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 depicted events, 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="587439204T" row="2" data_row="1"
colcount="1" col="2" col_id="2" name="claim_id" label="claim_id" type="string" index="IDX" just="I"
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="QIGCAAAAAAAAA=" 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="QNKSwAAAAAAAA=" 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="I" 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="I"
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="I"
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="QHmkAAAAAAAA=" 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 move the tagset indentation of the defined tagset.

```
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=xxx?" 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 (carriage return) to 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, 2011d, 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<sup>12</sup>. 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

<sup>12</sup> 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.

element being processed is the "claim\_id". If it is, then the value of "curr\_claim" is assessed to determine if the row 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;

```

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>
    </detail>
  </claim>
</claims>

```

## 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, and understand how to interact with web services that we might encounter in accessing our source data. 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.
- Chinthaka, E. (2007). Enable REST with Web services, Part 1: REST and Web services in WSDL 2.0. <https://www.ibm.com/developerworks/webservices/library/ws-rest1/>
- 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.
- Mack, C.E. (2010). Using Base SAS® to Talk to the Outside World: Consuming SOAP and REST Web Services Using SAS® 9.1 and the New Features of SAS® 9.2®. Proceedings of the SAS Global Forum 2010. Cary, NC: SAS Institute, Inc.
- Martell, C. (2008). SAS® XML Mapper to the Rescue. Proceedings of the SAS Global Forum 2008. Cary, NC: SAS Institute, Inc.
- Richardson, L. & Ruby, S. (2007). RESTful Web Services. Sebastopol, CA: O'Reilly.
- SAS Institute Inc. (2010). SAS® 9.2 XML LIBNAME Engine: User's Guide, Second Edition. 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® 9.2 Companion for Windows, Second Edition. Cary, NC: SAS Institute, Inc.
- SAS Institute Inc. (2011c). SAS® 9.3 Functions and CALL Routines: Reference. Cary, NC: SAS Institute, Inc.
- SAS Institute Inc. (2011d). SAS® 9.3 Output Delivery System: User's Guide. Cary, NC: SAS Institute, Inc.
- Schacherer, C.W. (2012). The FILENAME Statement: Interacting with the World Outside of SAS®. Proceedings of the SAS Global Forum 2012.
- Shoemaker, J.N. (2005). XML Primer for SAS® Programmers. . Proceedings of the 30<sup>th</sup> Annual SAS Users Group International Meeting. Cary, NC: SAS Institute, Inc.
- World Wide Web Consortium (2004). Web Services Glossary: W3C Working Group Note 11 February 2004. <http://www.w3.org/TR/2004/NOTE-ws-gloss-20040211/>
- 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/>

## **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](mailto:CSchacherer@cdms-llc.com)  
Web: [www.cdms-llc.com](http://www.cdms-llc.com)

SAS and all other SAS Institute Inc. product or service names are registered trademarks or trademarks of SAS Institute Inc. in the USA and other countries. ® indicates USA registration.

Other brand and product names are trademarks of their respective companies.