SESUG Paper 136-2020

Lazy Programmers write Self-Modifying code OR Dealing with XML file Ordinals

David B. Horvath, MS, CCP

ABSTRACT

The XML engine within SAS® is very powerful but it does convert every object into a SAS data set with generated keys to implement the parent/child relationships between these objects. Those keys (Ordinals in SAS speak) are guaranteed to be unique within a specific XML file. However, they restart at 1 with each file. When concatenating the individual tables together, those keys are no longer unique. We received an XML file with over 110 objects resulting in over 110 SAS data sets, which our internal customer wanted concatenated for multiple days. Rather than copying and pasting the code to handle this process 110+ times, and knowing that I would make mistakes along the way—and knowing that the objects would also change along the way—I created SAS code to create the SAS code to handle the XML. I consider myself a Lazy Programmer. As the classic "Real Programmers…" sheet tells us, Real Programmers are Lazy. This session reviews XML (briefly), SAS® XML Mapper, XML Engine, techniques for handing the Ordinals over multiple days, and finally discusses a technique for using SAS code to generate SAS code.

INTRODUCTION

XML, or eXtensible Markup Language is a method of creating files transferred between organizations. The primary advantage of using this form is that it is human readable and, more importantly, understandable. While other forms, like CSV are human readable, and to a certain extent, understandable when column headers are provided, they are a flat form. XML allows the creation of multiple levels of data with varying levels of cardinality. And it is certainly easier to deal with than the old method of creating a non-delimited fixed field size file.

With a CSV, if you want to transfer account information that could have multiple owners who each have multiple phone numbers, you have to define all those fields from the beginning. With XML, each account will contain one to many owners who will have one to many phone numbers. If there is only one account owner with one phone number, only that data will be sent, not empty fields.

While this is a nice feature it does make reading the data more difficult. An advantage and disadvantage is that you do not need to receive a record layout to begin reading and processing the data. The advantage is that you do not have to wait, the disadvantage is that this can lead to sloppiness and sudden, unexpected, changes (much like CSV).

SOME BACKGROUND ON XML

As I've already mentioned, XML stands for eXtensible Markup Language – it is a standard that was originally created in 1996 and consists of two main parts: Markup (the information about the data) and Content (the actual data). Markup is represented with tags; if you have ever looked at the source code for a web page, these will be familiar.

If you're lucky, the vendor will provide you with a definition or schema, known as the XSD: XML Schema Definition. If you are not, you'll just have the XML file itself. But you can work from that, especially with the tools that SAS has provided.

This also makes it easy for your data provider (internal or external to your organization) to change the layout: all they have to do is update the XSD and add the new data elements to the XML file. Or they can skip the XSD and just surprise you.

That flexibility can allow quick changes but if not communicated properly also to surprises.

The best way to learn about XML is to look at it:

```
<?xml version="1.0" encoding="UTF-8" standalone="no" ?><gpx
xmlns="http://www.topografix.com/GPX/1/1"
xmlns:gpxx="http://www.garmin.com/xmlschemas/GpxExtensions/v3"
xmlns:gpxtpx="http://www.garmin.com/xmlschemas/TrackPointExtension/v2"
creator="nÃ¼vi 2370" version="1.1"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.topografix.com/GPX/1/1
http://www.topografix.com/GPX/1/1/gpx.xsd
http://www.garmin.com/xmlschemas/TrackPointExtensionv2.xsd"><metadata><link
href="http://www.garmin.com"><text>Garmin
International</text></link><time>2012-04-12T04:31:39Z</time></metadata><wpt
lat="40.247249" lon="-
75.513001"><ele>28.72</ele><name>002</name><sym>Waypoint</sym></wpt><wpt
lat="39.764033" lon="-75.551346"><ele>61.17</ele>
```

But frankly, looking at the example from my portable navigation unit (“GPS”) isn’t very helpful in this format. If you look at a raw XML file in a text editor, this is what you will see.

Fortunately, web browsers will display XML for you with formatting and indenting that is not included in the original file. This helps to identify parent/child relationships. In this case, we’re looking at GPX or “GPS Exchange Format” which is designed as a common data format for the exchange of waypoints (locations), routes (how to get somewhere), and tracks (how you actually got there).

This makes it easy to process this kind of data in general. For this paper, it means I have easily available examples that do not violate the intellectual property and proprietary information of my employer.

Error! Reference source not found. shows XML formatted in a much more readable, pretty format.

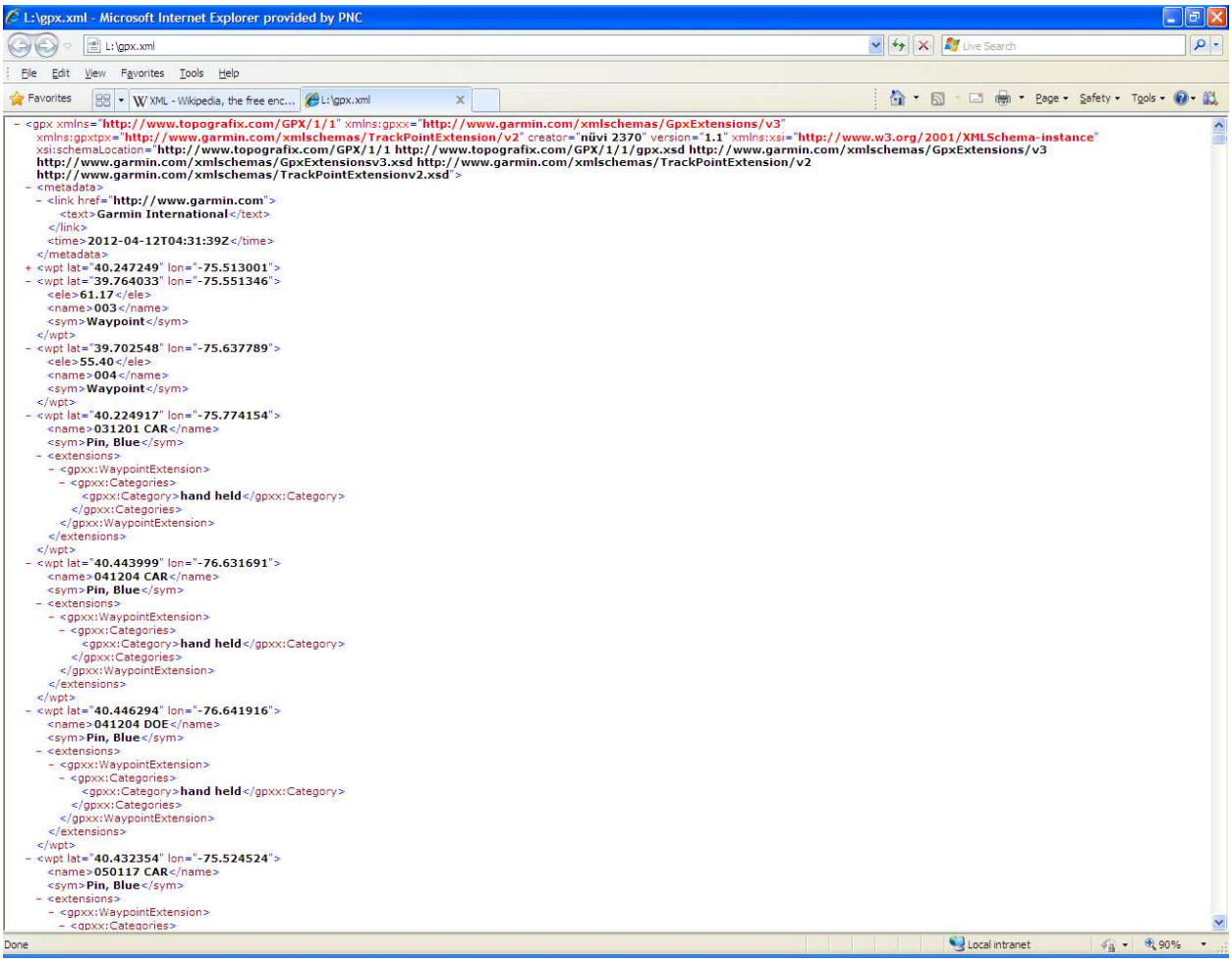

Figure 1. XML Viewed in a Web Browser

For the remainder of the paper, we will reference this example:

```
  <?xml version="1.0" encoding="UTF-8" standalone="no" ?>
<gpx xmlns="http://www.topografix.com/GPX/1/1"
xmlns:gpxx="http://www.garmin.com/xmlschemas/GpxExtensions/v3"
xmlns:gpxtpx="http://www.garmin.com/xmlschemas/TrackPointExtension/v2"
creator="nuvi 2370" version="1.1"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.topografix.com/GPX/1/1
http://www.topografix.com/GPX/1/1/gpx.xsd
http://www.garmin.com/xmlschemas/GpxExtensions/v3
http://www.garmin.com/xmlschemas/GpxExtensionsv3.xsd
http://www.garmin.com/xmlschemas/TrackPointExtension/v2
http://www.garmin.com/xmlschemas/TrackPointExtensionv2.xsd">
 <metadata>
 <link href="http://www.garmin.com">
  <text>Garmin International</text>
  </link>
  <time>2012-04-12T04:31:39Z</time>
  </metadata>
```

```
<wpt lat="40.224917" lon="-75.774154">
 <name>031201 CAR</name>
 <sym>Pin, Blue</sym>
<extensions>
<gpxx:WaypointExtension>
<gpxx:Categories>
 <gpxx:Category>hand held</gpxx:Category>
 </gpxx:Categories>
 </gpxx:WaypointExtension>
 </extensions>
 </wpt>

<trk>
 <name>Active Log: 11 APR 2012 10:17</name>
<trkseg>
<trkpt lat="57.722606" lon="11.846760">
 <ele>-18.61</ele>
 <time>2012-04-11T08:17:40Z</time>
<extensions>
<gpxtpx:TrackPointExtension>
 <gpxtpx:course>358.59</gpxtpx:course>
 </gpxtpx:TrackPointExtension>
 </extensions>
 </trkpt>
 </trkseg>
</trk>

</gpx>
```

Elements like <trk> (track) can have repeating sub-elements like <trkseg> (track segment) and <trkpt> (track point). A track point is a bread crumb along the way of your trip. Before I look at the data itself, I don't know how many tracks we will receive, how many segments each will have, nor how many bread crumbs were dropped along the way. I may not even know the data model for this information.

Obviously, working with GPX, there are plenty of sources of information, so in this context the example is silly. But proprietary data from a vendor or less than cooperative internal data provider, you may not. The GPX XSD (XML Schema Definition) is available for download – describing exactly what to expect in the XML data file.

The purpose of this paper is not to teach you XML coding; there are plenty of sources for that starting with http://en.wikipedia.org/wiki/Xml but having this background will help you when dealing with actual data.

SAS XML MAPPING TOOL

The SAS XML engine uses its own format for describing the contents of an XML file known as a "map". The map goes beyond the definitions you might find in an XSD because it defines how you want to present the data in SAS datasets including how to connect between the various structures.

To make your life easier, SAS provides a tool to create this map available as a free download from http://support.sas.com/kb/33/584.html

The SAS XML Mapper can process the XML data file itself or an XSD to create the required map. When working with XML itself, it parses the file much like proc import. And, like proc import, it will look at a subset of a large file. As a result, any element it does not encounter, or if there are larger text fields later in the file, it may not cover all the elements correctly. Much like proc import, it also has to "guess" at the data type of elements.

A better approach is to use XML Mapper to process an XSD file when creating the map. The XSD provides a full definition reducing the "guessing" the tool has to perform. The only downside is that an XSD is not always available.

A nice feature of the tool is that it creates its own keys to connect the elements – known as "ordinals". A child element will contain the ordinal of the parent. These ordinals are unique surrogate keys assigned during processing by the XML engine.

You also edit the map (which is in a form of XML) yourself if you'd like. For instance, adding additional ordinals allowing grandchildren to connect directly to the top level. In the Account/Owner/Phone number example, Account will contain its own ordinals, Owner will have both its and the Account ordinals, Phone will have both its and the Owner ordinals. But if you want to connect Phone to the account directory, you can manually edit the file and add them.

The map itself is in XML format

SAS XML ENGINE: CODING TO ACCESS XML DATA

The coding within your SAS program to access XML data, once the map is created, is really very simple. First you need to define the files necessary:

```
filename  test "/export/home/myid/gpx.xml";
filename  SXLEMAP "/export/home/myid/gpx.map";
libname   test xml xmlmap=SXLEMAP access=READONLY;
```

The filename for your XML file and the libname should match ("test") in this example. You also need to define the name of the map itself.

Once that is complete, you can use the libname like any other:

```
proc contents data=test._all_ ;
run;
```

Or read any of the member elements (treated like SAS datasets) almost like any other:

```
data tableonly;
   set test.member END=EOF;
   output;
run;
```

Writing to an XML dataset (even without the "access=READONLY" parameter) is a bit more difficult:

```
ERROR: XMLMap= has been specified on the XML Libname assignment. The output
produced via this option will change in upcoming releases. Correct the XML
Libname(remove XMLMap= option) and resubmit. Output generation aborted.
```

Much like most of the SAS capabilities, everything you ever wanted to know about the SAS XML engine is available at http://support.sas.com/rnd/base/xmlengine/index.html

OUR PROBLEM

The reason behind this discussion of how SAS handles XML files came about from an internal request to process a vendor produced XML file. As is often the case, there was limited documentation and a short timeline. The vendor had already been contracted and data delivery beginning shortly. As a result, we had limited time to prepare and needed a flexible solution. And, unfortunately, there was no internal experience with these tools.

Complicating the situation was the contents of the XML: literally hundreds of internal "objects" (in my Account/Owner/Phone number, each of those is an "object"). Each of those objects maps to one dataset.

While the map can be manipulated to combine multiple XML objects into one dataset, at this point we did not know enough about the data itself to be able to make those decisions. We needed to process the data into a usable form (datasets) to be able to see and learn about it.

Ultimately, this was to be a daily file running through an enterprise scheduler (“hands off” – no manual intervention) producing a full history over time.

DEALING WITH ORDINALS OVER MULTIPLE FILES

The catch with the ordinal fields used to connect the various objects is that they are only unique to a specific XML data file, not over time. Each day, the first record in each object has the ordinal value of one. But in order to build history, I need uniqueness over time. In appending today’s data to yesterday’s to build my history, the Ordinals need to change.

There really is a simple solution: find yesterday’s maximum Ordinal value for each table (either from history or a saved table; history is easier to maintain), add that maximum to today’s generated values, and append the resulting “today” table to the existing history.

A better approach would be to perform the analysis to build exactly the records you need in the form you need. But you have to take the time to understand the data to perform the analysis. Time we did not have.

Shifting back to the GPX example, we can look at the proc contents for two of the objects: gpx (the main parent) and rte (route, one of the children).

Error! Reference source not found. shows the contents of the gpx Object – including the Ordinal unique key.

Variable	Type	Len	Format	Informat
creator	Char	32	$32.	$32.
extensions	Char	32	$32.	$32.
gpx	Char	32	$32.	$32.
gpx_ORDINAL	Num	8	F8.	F8.
version	Char	32	$32.	$32.

Table 1. gpx Object

Error! Reference source not found. shows the contents of the rte (route) Object which includes its Ordinal unique key as well as that of the parent.

Variable	Type	Len	Format	Informat
cmt	Char	32	$32.	$32.
desc	Char	32	$32.	$32.
extensions	Char	32	$32.	$32.
gpx_ORDINAL	Num	8	F8.	F8.
name	Char	32	$32.	$32.
number	Num	8	F8.	F8.
rte	Char	32	$32.	$32.
rte_ORDINAL	Num	8	F8.	F8.
src	Char	32	$32.	$32.

type	Char	32	$32.	$32.

Table 2. rte (route) Object

These are only two of the total of 19 objects (XML pseudo Datasets) in a GPX file that would result in creating essentially the same code to perform the Ordinal manipulation 19 times. While it is minor pain when dealing with 19 sets, can you imagine doing that for the hundreds involved with our Vendor's XML?

I can't. I'd admit it: I'm lazy and I make mistakes. I'd really rather not copy & paste & edit the same code 19 (or 190) times. And I'd rather not have to repeat the process every time the file changes (or a new element appears; remember that we had no XSD).

Since the same process applies for every one of the elements, why not let code do the work for me?

USING SAS CODE TO GENERATE SAS CODE

The mechanism to create self-modifying code within SAS is rather simple since it is an interpreted language. You use File, Put, and %include:

```
filename sourcecd "gpxxml_generated&DATADATE..sas";

data _null_;
    file sourcec2;
    set temp.maxvals end=EOF;
/* use put statements */
run;

%include sourcec2;
```

The maxvals dataset contains the maximum ordinals from yesterday. I create a file that contains SAS code, and by including it, cause execution of that new code. In detail:

```
filename sourcec2 "xml_generated&DATADATE._2.sas";

data _null_;
    file sourcec2;
    set temp.maxvals end=EOF;

    if (_n_ = 1) then do;
          put "libname temp 'temp';";
    end;

    put "data temp." tableonly "; set " member "END=EOF;";
    if prikey NOT = "" AND prival NOT = . then
          put prikey" = " prikey " + " prival";";
    if parkey NOT = "" AND parval NOT = . then
          put parkey" = " parkey " + " parval";";
    put "output;";
    put "run;";

    put "proc datasets; append base=output." tableonly
          " data=temp." tableonly "; run;";

    if EOF then do;
          put " run;";
    end;
```

```
run;
```

This code will generate the following code that is included back into this program. I also could have created a full program and executed with a new sas command. The resulting code snippet defined in sourcec2:

```
/* Libname test and output defined in main program */
libname temp '/export/home/myid/temp';
data temp.AUTHOR ;
    set test.AUTHOR END=EOF;
    METADATA_ORDINAL  = METADATA_ORDINAL  + 257 ;
    output;
run;
proc datasets; append base=output.AUTHOR  data=temp.AUTHOR ;
run;

data temp.BOUNDS ;
    set test.BOUNDS END=EOF;
    METADATA_ORDINAL  = METADATA_ORDINAL  + 257 ;
    output;
run;
proc datasets; append base=output.BOUNDS  data=temp.BOUNDS ;
run;
```

This code is executed by including it back into the main program. Each day, the generated code is a little different because the maximum ordinals (in the example above, the value 257) changes each day. That way, the history contains unique ordinals over time.

AFTERTHOUGHTS

While I could have merged the maximum ordinal into each record or used macro values for that maximum ordinal, this approach means I am only processing each file the least number of times. This is important when dealing with over 200 objects from an XML file that occupies over 6 Gigabytes (on a daily basis). The fewer passes through the data, the better.

We encountered issues dealing with the raw files as well. In one case we encountered a bad tag that broke the XML engine. In order to remove it we had to resort to UNIX tools (one line of awk code to strip out the tag). We did explore the possibility of editing the file within SAS as a plain text file, having data records that exceeded 88,000 bytes caused difficulties. Although _infile_ was automatically set, it was limited to 32,767 bytes. Trying to read each record into three 32K character fields (with the possibility that the maximum record length could grow) and finding that string that could cross those fields would have required rather complex coding. By going outside the box, processing the 6 Gigabyte file took mere minutes.

One disadvantage of the XML engine is that each object we want to process (convert to a dataset) requires the full XML file to be parsed. As a result, we are parsing that same file about 200 times. Because of the elapsed time and computational intensity of the parsing, the code was changed to create 10 rsubmit packages to take advantage of the multiple CPU on our servers. Although this imposed a high I/O load on the server, it dramatically reduced the elapsed time required.

Looking back, it might have been better to save off the maximum ordinal for each object at the end of each run rather than getting it from the history. My concern was that it would be difficult to fall back if a rerun was needed and the difficulty in adding new objects in the future.

In order to dynamically determine the objects (the resulting datasets) and the variable names for ordinals, I used proc contents. I could have used the SQL dictionary tables but was not familiar with them at the time.

Although these examples were written with version 1 of the XML engine, we moved to version 2, the only code change that was required was changing the libname:

```
libname test xmlv2 xmlmap=SXLEMAP access=READONLY;
```

Of course, we did have to change the map to the new syntax but the new XML Mapper took care of most of that effort.

I also noticed that the set nobs= statement did not work with XML. It compiles (no warning or error) and executes, but returns the value zero. I learned that while working on a presentation for PhilaSUG on NOBS.

CONCLUSION

The XML engine is a powerful tool for parsing these files. But it comes at the cost of having to parse large files multiple times. Dynamic code helps to minimize the computational cost while enhancing flexibility.

ACKNOWLEDGMENTS

I want to thank the organizers of this great conference, my employer for their willingness to allow me to expand my horizons through events like this, and, last but certainly not least, my spouse Mary who doesn't complain when I spend so much time at the keyboard working on documents like this.

CONTACT INFORMATION

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

David B. Horvath, CCP
+1-610-859-8826
dhorvath@cobs.com
http://www.cobs.com
LinkedIn: https://www.linkedin.com/in/dbhorvath/