Cascade Solutions
SUCCESS DELIVERED

18312 West Creek Drive
Tinley Park, IL 60477
P: 708.802.6000 F: 708.802.6008
www.cascadesolutions.com

The ABCs of XML

By John T. Sever
Cascade Controls, Inc.

The project was nearly finished. The end was in sight after an incredible effort of long hours, weekends, battles, changes, and all of the normal challenges that accompany a large project. We had delivered roughly 80% of the automation software for a life sciences S88 batching application when a problem surfaced - a big problem. Equipment modules and phases included commands to devices that could cross controller boundaries. The code construct empoloyed for these commands were non-deterministic when crossing controller boundaries – by design. This meant that the destination may not receive the command. The problem appeared sporadically but rarely on our development system and we could never recreate the issue in a controlled environment. On a large plant-wide system with large IO bus networks, the problem was suddenly pervasive.

The code had to be changed. To change one instance was simple enough but we didn't know how many instances existed throughout the nearly 2,000 phases and equipment modules each containing many steps which in turn contain multiple actions. My initial ballpark estimate to manually modify and validate the changes was between one and two manyears. Of course the project didn't have money or time burning a hole in its proverbial pocket nor were qualified resources available to do the work.

A few years earlier this would have been a disaster for everyone involved. A few years earlier, we didn't know what was possible with XML and XSLT.

Once we understood the problem and the required code changes, we knew that only XSLT could help us make the necessary code modifications with minimal project impact. We created an XSLT transform to find every action with the faulty code construct, modify the code appropriately, and record each code modification in a change log file. The code was exported from the DCS system as structured text, converted to XML, transformed with XSLT to fix the problems, converted back to structured text, and imported back into the system. The XSLT transform was validated so that each change did not have to be individually tested. All of the changes were completed with less than a man-month of work, and no impact to schedule, and minimal budget impact.

Many in the world of automation have not bothered to learn about XML (eXtensible Markup Language) or XSLT (eXtensible Stylesheet Language Transformation) believing they are technologies for web developers, IT personnel, and consultants. The truth is that XML and XSLT are finding their way into nearly aspect of computer technology today. XML based technology has spread so pervasively and so widely across application domains that many are unaware of its impact in their own discipline or of the potential value it brings.

This four part article will introduce the basics of XML, introduce XSLT as a query and transformation language for XML data, and provide examples of how these technologies can be used in the world of automation. The three parts to follow are summarized below:

Introduction to XML

XML is a flexible general purpose data storage structure in human readable text format. Despite its name, XML is not really a "language" like HTML. Instead, XML provides a set of rules for anyone to create their own markup language. Anyone familiar with HTML will recognize many similarities with XML. These similarities made it easy for web developers to jump aboard the XML bandwagon. Its proliferation on the web has created some misconception of XML as a web-only technology. In fact Dictionary.com defines XML as follows:

> *A metalanguage written in SGML that allows one to design a markup language, used to allow for the easy interchange of documents on the World Wide Web.*

XML has been used for much more than the easy interchange of document on the World Wide Web. A more appropriate definition might read:

> *A metalanguage written in SGML that allows one to design a markup language, used to allow for the easy interchange of data.*

In this article I will introduce the basic rules of XML, terminology, and related standards. You'll learn how to read XML documents and how to create your own XML documents. The article will also discuss declarations, processing instructions, namespaces, well-formedness, validity, DTDs, schemas, and encoding. Finally, I'll point you to a number of resources and applications to help you learn more about XML and to help you work with your own XML documents.

Introduction to XSLT

I recently overheard the following conversation between two automation engineers:

Engineer 1 - They want to review the recipes.

Engineer 2 - How do they want them?

Engineer 1 - Can you print them out?

Engineer 2 - That'll take a few days.

Engineer 1 - Can you export them?

Engineer 2 - I can save them as XML.

Engineer 1 - What do I do with that?

Engineer 2 - I don't know.

I'm not sure how this problem was resolved because I couldn't stick around to help them out. Had either of them known about XSLT, they could have transformed those XML recipes into any number of formats. The easiest of these would have been a CSV that could be opened in Excel. Another option (my favorite) would be to transform the recipes into MS Visio drawings. Either of these two tasks could have been accomplished in a few short hours and it would have worked on all 1,500 recipes.

XSLT is a standard for converting XML to something else. The something else is growing rapidly. Most frequently, the something else is XML in a different data structure. This article opened with a problem whose solution was to convert one XML file (with bad code) to another XML file (with good code) Other common formats created by XSLT include HTML and plain text. One of the most popular formats our company creates with XSLT is MS Word documents. Microsoft supports XML as a format for most or all of its Office 2003 applications.

In this article I will introduce XSLT as a declarative language. I will describe the transformation process and explain some of the basic XSLT programming elements. The article will also discuss XSLT

expressions, match patterns, recursion, templates, default templates, XSLT processors, scripting, and some of the limitations of XSLT. Finally, I'll point you to a number of resources and applications to help you learn more about XSLT and to get up the XSLT learning curve.

Putting It All Together

Learning a new language is not a trivial undertaking. There needs to be a big pay-off if you're going to spend the time to learn new data structures and languages. Rarely has a new language made your life better. This one will.

The final article will provide real world examples of what can be accomplished with XML and XSLT. I'll make sure to point out obstacles that commonly arise and how to over come these. I'll provide XML samples from real automation systems and accomplish tasks that have practical use in the everyday life of an automation engineer.

I hope that once we've finished this series you'll be well armed and ready apply these concepts to your everyday tasks.

Online Resources

www.w3c.org

www.xml.com

www.xml.org

www.topxml.com

msdn.microsoft.com/xml

John T. Sever is president and founder of Cascade Controls, Inc., and has 20 years of experience as a process and automation engineer. He can be reached at 708/802-6000 or ,at johnsever@cascon.com.

Cascade Solutions
SUCCESS DELIVERED

18312 West Creek Drive
Tinley Park, IL 60477
P: 708.802.6000 F: 708.802.6008
www.cascadesolutions.com

The ABCs of XML
Part 2 - XML

By John T. Sever
Cascade Controls, Inc.

Introduction

The World Wide Web Consortium's (www.W3C.org) XML Recommendation opens with a list of ten design goals. The first goal states *XML shall be straightforwardly usable over the Internet.* Straightforwardly or not, XML **is** used extensively over the Internet and in fact has become a defacto standard for data interchange. Because of its association with Internet applications, automation engineers have sidestepped XML assuming it has little applicability to their daily work. This is a mistake. Although this technology has been used extensively for Internet based applications, XML is an extremely simple and flexible data format with untold uses waiting to be uncovered in the world of industrial automation.

Alone, XML data is simply raw text that has little to offer an automation engineer. But XML is not alone. Developers everywhere have jumped aboard the XML bandwagon to create a seemingly bottomless reservoir of tools, applications, services, and standards all designed to create, consume, translate, store, and present XML data. This infrastructure of supporting applications is what makes XML such a compelling choice for application data. This article will introduce XML fundamental concepts for those who have so far managed to avoid this important technology. Parts 3 and 4 of this 4 part article will address XML supporting technologies.

Not A Typical Language

XML is not a language in the sense that there are defined keywords, functions, or statements. XML is often compared to HTML because it works well with HTML applications, has similar markup, and has been joined with HTML to create the XHTML specification. However, the HTML specification defines a list element tags like `<body>`, `<h1>`, `<b>`, and `<i>` with defined behavior for HTML browsers. XML lacks a defined set of tags and allows anyone to create their own set of tags and attributes to suit their own application needs. Instead, the XML specification defines a set of markup rules that must be followed for the marked up text to be interpreted as XML data.

Document Metaphor

XML is organized in a logical or physical structure called a *document*. An XML document may be a file on disk, it may be streamed from a server, or it may be hard coded text inside an HMI VBA application. Though the data may have many different sources, the document metaphor still applies as long as it is *well-formed*. To be well-formed means that the document adheres to all of the markup rules defined in the XML recommendation.

Sample XML

The sample fragment was lifted from a recipe exported from Rockwell's RSBatch product. Rockwell defined the element and attribute names for describing an RSBatch recipe. Other system vendors may define a separate set of elements and attributes to describe a batch recipe. If you are a system integrator that works with batch recipe software from different system vendors, you may choose to define your own system agnostic batch recipe XML data structure (or *schema*) for internal development purposes that can easily be converted to/from a vendor specific structure.

```
<!-- This is an XML comment -->
<Step XPos="600" YPos="600" AcquireUnit="true">
  <Name>FEED:1</Name>
  <StepRecipeID>FEED</StepRecipeID>
  <UnitAlias>UNIT500</UnitAlias>
  <FormulaValue>
    <Name>AGIT_SPEED_SP</Name>
    <Display>true</Display>
    <Value />
    <Real>75</Real>
    <EngineeringUnits>RPM</EngineeringUnits>
  </FormulaValue>
</Step>
```

The first thing to notice is the angle brackets (< and >) which mark an XML *tag*. An XML *element* includes a start tag like `<Step>`, an end tag like `</Step>`, and everything between the two. Notice that the `Step` element contains four child elements `Name`, `StepRecipeID`, `UnitAlias`, and `FormulaValue`. The `FormulaValue` element contains 5 child elements. This parent/child relationship between elements shows that XML can support hierarchical data structures.

XML is designed to be self-describing. A document's data is stored within element values and attribute values such that element and attribute names describe the data they hold much like data is described in a relational database by table names and field or column names.

Data stored as an element value is the text between start and end tags. In our sample, the value for the element `EngineeringUnits` is `RPM`. Data stored as an attribute value is found on the quoted right side of a Name="Value" pair. In our example, the `Step` element contains three attributes named `XPos`, `YPos`, and `AcquireUnit` which have attribute values `600`, `600`, and `true` respectively.

To be considered a well formed, an XML document must adhere to the constraints of well-formedness defined in the W3C XML specification. I've distilled these constraints into the following 10 easy rules.

1. **XML is just plain old text**

   XML is designed to be human readable as text. This means that any text editor can be used with an XML document. A simple text editor will treat an XML document just as it would an INI file, a CSV file, or any text file.

2. **XML is Data**

   XML is designed as a flexible self-describing data structure. By itself, XML cannot do anything nor does it define how data should be processed or handled. By contrast, HTML includes both data and a description of how it should be displayed in a browser.

3. **XML documents must have one root element**

   There can be only one top level root element in an XML document and all other elements must be between the root element start and end tags.

4. **XML white space data is preserved**

   HTML reduces consecutive white space characters to a single space character. With XML, white space is interpreted as data – just as any other character.

5. **XML naming rules**

   Element names cannot include white space, must start with a letter and cannot include characters that are used for markup such as <, >, ;, &, among others. It is generally a good idea to limit element and attribute names to letters, numbers, and underscore.

6. **XML elements must be closed**

   An element can be closed with an end tag or optionally with the short hand notation for empty elements. By contrast, HTML does not require that elements be closed. In fact most browsers will attempt to render any HTML element whether or not it is closed properly – XML is not so forgiving.

   An empty element is one with no value and no child elements (although it may have attributes). An empty element may be closed with the short hand notation /> at the end of the start tag. For example <Value /> is equivalent to <Value></Value>.

7. **XML elements must be properly nested**

   In HTML, elements were allowed to overlap like this <i>bold and italic text italic only</i>. This type of element crossing is not allowed. An element that starts inside a parent element must end inside the same parent before the parent element is closed.

8. **XML is case sensitive**

   An XML tag <recipe> is not the same as <Recipe>. HTML however is not case sensitive so <h1> is identical to <H1>.

9. **XML attribute names must be unique within an element**

   An element may have any number of attributes but each attribute name must be unique. The following example is incorrect:

```
<People Person="John Doe" Person="John Smith" />
```

   This example could be structured properly as follows.

```
<People>
   <Person>John Doe</Person>
   <Person>John Smith</Person>
</People>
```

   or

```
<People>
   <Person Name="John Doe"/>
   <Person Name="John Smith"/>
</People>
```

10. **XML attribute values must be quoted**

    XML attribute values must be enclosed either in single quotes or double quotes. If an attribute value contains a double quote character enclose the value in single quotes. Likewise, if the attribute value contains a single quote, enclose the value in double quotes. For attribute values that may include either type of quotation character, standard HTML character entities may be used – " for the double quote character and ' for the single quote character.

Comments

The markup for a comment in XML is identical to HTML. The comment opens with `<!--` and closes with `-->` and may span multiple lines.

XML Declaration

An XML document may begin with an optional XML declaration. The XML declaration must precede all other content is not considered part of the XML document. It is used to provide information to XML processors about the document's content. Because the declaration is not an element it must not have a closing tag. The declaration looks like this

```
<?xml version="1.0" encoding="utf-8" standalone="yes"?>
```

If the declaration is included, version is the only required attribute and must have a value of either `1.0` or `1.1`. Version 1.1 supports special Unicode character handling functionality that is rarely needed and therefore version 1.0 is used almost exclusively.

The encoding attribute defines the character encoding used by the document so that an XML processor may properly parse the document. The default encoding used by XML processors is UTF-8.

The `standalone` attribute is used when the document refers to another documents such as a schema document (described later). When not specified, this attribute defaults to "yes" (no reference document).

Processing Instructions

Any number of processing instructions may appear below the XML declaration and before the root element. It must be enclosed in <? and ?> like the XML declaration and provides application specific handling information. A Microsoft Word 2003 XML document may include the following processing instruction which tells the Windows operating system to identify the XML document as an MS Word file. When double-clicked, an XML file with this processing instruction will open in MS Word.

```
<?mso-application progid="Word.Document"?>
```

XML Validation

An XML document has a specific structure of element names, attribute names, and hierarchical parent-child relationships. As long as a document meets the requirements for *well-formedness* it can have any structure and contain any data. This flexibility is what makes XML extensible. However, applications that interpret XML documents have expectations that the XML will adhere to a particular structure. Validation is the process of checking an XML document for conformance to a defined structure or *schema*. A schema can be defined within the XML document or a reference can point to an external schema document. There are multiple standards for defining a schema including DTD – Document Type Definition, XSD – XML Schema Definition Language, and XDR – XML Data Reduced. An XML document that that adheres to a defined schema definition is said to be *valid*.

A schema is not required when developing XML applications and in fact can significantly complicate XML application development. When you control a document's content and related applications, you can work more efficiently without a schema.

Software vendors that support XML data normally publish a schema so that other applications can properly validate content before working with a document. A control system vendor that supports import of XML data into the control system will likely validate a document before the import process to prevent loading data that may lead to a control system fault.

Namespaces

XML namespaces solve a problem that can occur when an element name may have different meaning within a single document. For example, the element name `template` is an XSLT keyword with meaning

different than the `template` element used in an MS Word XML document. All elements and attributes in an XML document are included in a namespace even if a namespace is not explicitly declared. When no namespace is defined in a document, content is included in the default null namespace.

A namespace may be defined as an attribute of the start tag of an element with the following format: `xmlns:prefix="namespaceURI"`

Where a namespace is declared for an element, all child elements with the same prefix are included in the same namespace. The element where the namespace is declared may also be included in the namespace if the same prefix is used in the element name.

```
<cc:Recipe xmlns:cc="http://www.cascon.com/Recipe">
   <cc:Step cc:XPos="600" cc:YPos="600" AcquireUnit="yes">
          <cc:Name>Feed:1</cc:Name>
          <UnitAlias>Unit500</UnitAlias>
   </cc:Step>
</cc:Recipe>
```

In the sample above, the prefix `cc` refers to the namespace `http://www.cascon.com/Recipe`. Elements included in this namespace include Recipe, Step, and Name. The element UnitAlias and the attribute AcquireUnit are included in the default null namespace.

The namespace prefix `cc` serves as a shorthand or alias notation for the full namespace `http://www.cascon.com/Recipe`. The actual namespace may be any string value but it is meant to be globally unique. XML parsers do not enforce uniqueness nor do they expect any particular notation such as a web URI. A web style URI (Universal Resource Identifier) is frequently used because a real web URI like www.cascon.com is guaranteed to be globally unique across the internet – greatly minimizing the chance of colliding namespaces.

To simplify this example, the namespace can be declared as the default namespace with no prefix as shown in the following example.

```
<Recipe xmlns="http://www.cascon.com/Recipe">
   <Step XPos="600" YPos="600" AcquireUnit="yes">
          <Name>Feed:1</Name>
          <UnitAlias>Unit500</UnitAlias>
   </Step>
</Recipe>
```

Notice that the namespace attribute `xmlns` no longer includes the prefix definition `cc`. Without a prefix, a namespace becomes the default namespace for the element where it is declared. This makes the Recipe element and all its child element members of the namespace `http://www.cascon.com/Recipe`. Default namespaces apply to elements only, not to attributes. Therefore, the attributes of the Step element are included in a null namespace (equivalent to `xmlns=""`), not the default namespace. This special behavior for attributes can be quite confusing. This quirk of default namespaces is not difficult to work around as long as you understand how it works.

You'll likely not need to bother with namespaces in documents created for internal purposes. However, you'll need to understand namespaces when working with vendor generated XML files. You will see the importance of namespaces in next week's XSLT article.

The ABCs of XML
Part 3 - XSLT

By John T. Sever
Cascade Controls, Inc.
18312 South West Creek Drive
Tinley Park, IL 60477

Introduction

XML data is increasingly more prevalent in the world of industrial automation. System vendors increasing provide options for data interchange through XML and industry organizations adopt XML as a standard for data transport like the OPC Foundation's OPC XML-DA specification. Still, many front line automation engineers find no benefit to old data restructured in a new format as XML. They are correct that little benefit is found in XML alone. Instead the benefits they seek may be found in technologies designed to support, consume, and manipulate XML data. One such technology from the World Wide Web Consortium (www.w3c.org) is a language specifically designed for transforming or restructuring an XML document into something else. The eXtensible Stylesheet Language Transformations (XSLT) is a simple yet powerful language that can bring enormous benefit to those willing to learn it.

In this article, I will introduce the basics of XSLT through an example derived from an actual customer request. My coverage of the topic will be limited by the space available in this format however, I hope to awaken interest in a technology that might otherwise be overlooked and thereby give to some, a hint of the benefits waiting to be uncovered.

The Project

My customer maintains a large DeltaV automation system in an FDA qualified environment. The customer would like a tool that simplifies daily demands of managing a large configuration. Such a tool, we have determined, is most easily developed around a relational database. Therefore we must transfer the configuration from control system to database in a structure that best supports the requirements of this tool.

A prototype of this tool is under development using Microsoft Access 2003 because of its built-in features for reporting and rapid application development. One such feature is the ability to import XML data through the File menu (File/Get External Data/Import) allowing the application of an XSLT transform before importing. This feature imports element normal XML data by creating new tables with structures based upon structures found in the XML document. The user may optionally choose to append data to existing tables following a set of mapping rules that match table and field names to element names in the XML document.

The Transformation

Unfortunately, the XML structure of a DeltaV configuration does not match the desired data structures to meet our analysis requirements and so the raw XML data must be restructured or transformed into a different XML structure better suited to the customer's requirements. Before learning the benefits of XSLT I would have solved this problem by writing VBA code to load the XML document and programmatically walk the tree of nodes adding new records to database tables using either the DAO or

ADO object models. The XSLT solution presented here requires less code, is more robust, easier to develop, easier to maintain and modify, and is significantly faster than the VBA solution.

The XSLT stylesheet shown here restructures the XML for importing control modules and their parameters. By dissecting this stylesheet I will explain basic XSLT concepts while explaining solutions to common data structure problems.

Before reading on, get familiar with the basic structure of a DeltaV XML file. Notice that each control module is defined in a `<module>` element that has attributes for tag, plant_area, category, user, and time. Also review the <attribute> and <attribute_instance> elements that are children of a <module> element.

```
<?xml version="1.0" encoding="utf-8"?>
<xsl:stylesheet version="1.0"xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:output encoding="utf-8" indent="yes"/>

  <xsl:template match="/fhx">
     <root>
        <xsl:apply-templates select="module">
           <xsl:sort select="@tag" data-type="text" order="ascending"/>
        </xsl:apply-templates>
     </root>
  </xsl:template>

  <xsl:template match="module">
     <xsl:copy>
        <xsl:copy-of select="description|period|controller|type"/>
        <xsl:apply-templates select="@*"/>
        <xsl:apply-templates select="attribute"/>
     </xsl:copy>
  </xsl:template>

  <xsl:template match="module/@*">
     <xsl:element name="{name()}">
        <xsl:value-of select="."/>
     </xsl:element>
  </xsl:template>

  <xsl:template match="module/attribute">
     <xsl:variable name="Instance" select="../attribute_instance[@name=current()/@name]"/>
     <Parameter>
        <ModuleTag><xsl:value-of select="../@tag"/></ModuleTag>
        <name><xsl:value-of select="@name"/></name>
        <type><xsl:value-of select="@type"/></type>
        <xsl:copy-of select="group|connection"/>
        <xsl:copy-of select="rectangle/*"/>
        <categories>
           <xsl:for-each select="category">
              <xsl:value-of select="category"/>
              <xsl:if test="position()!=last()">,</xsl:if>
           </xsl:for-each>
        </categories>
        <enum>
           <xsl:value-of select="$Instance/value/set"/>
        </enum>
        <value>
        <xsl:choose>
           <xsl:when test="@type='BOOLEAN' or @type='FLOAT' or @type='INT16' or
                          @type='UINT16' or @type='UINT8' or @type='UNICODE_STRING' or
                          contains(@type, '_WITH_STATUS')">

              <xsl:value-of select="$Instance/value/cv"/>
           </xsl:when>
           <xsl:when test="@type='ENUMERATION_VALUE'">
              <xsl:value-of select="$Instance/value/string_value"/>
           </xsl:when>
```

```
          <xsl:when test="@type='EXTERNAL_REFERENCE' or @type='INTERNAL_REFERENCE'">
            <xsl:value-of select="$Instance/value/ref"/>
          </xsl:when>
        </xsl:choose>
        </value>
      </Parameter>
    </xsl:template>
</xsl:stylesheet>
```

Stylesheet as Transformation

The top element in an XSLT transformation must be the `<xsl:stylesheet>` element and it must include a version attribute and the namespace declaration shown here. An XSLT transformation is also an XML document and therefore it must have a single top level element `<xsl:stylesheet>`. This element must include the namespace `http://www.w3.org/1999/XSL/Transform` and it must have a version attribute. By convention, the namespace is generally assigned the `xsl:` prefix used throughout the document to distinguish XSLT language elements from literal output.

XSLT Transformation Processors

Execution of a transformation is performed by an XSLT processor that accepts XML and XSLT documents as input and generates a document as output. Output is created through execution of *templates* declared within the XSL document. The processor walks the XML tree matching each node to a template in the XSLT document. If the processor cannot find a matching template in the XSLT file, it executes a built-in template inside the processor.

The built-in template for element nodes selects child nodes of the current element and tries to find templates that match these children. This built-in behavior results in automatic recursion down every branch of the XML source document tree. There is also a built-in template for text nodes which copies the text to the output. The effect of this built-in template is usually unexpected and unwanted but can be easily prevented if you understand this behavior.

When a matching template is found for a given node (not a built-in template) the processor will not automatically recurse its children effectively ending the transformation process for all descendent nodes. The developer may programmatically choose to continue processing child nodes with <xsl:for-each> or by invoking <xsl:apply-templates> providing fine control of how child nodes are processed.

Output Formatting

The second line of our transform, `<xsl:output>`, controls output format and must appear before any templates. The attributes specify output formatting instructions for the processor.

```
  <xsl:output encoding="utf-8" indent="yes"/>
```

Templates

In our example, the processor starts with the document root and, finding no matching templates, invokes the built-in template which selects children of the document root and looks again for matching templates. The only child of the document root is the <fhx> element which is passed to our first template that matches <fhx> elements as children of the document root (identified by the slash).

This template generates a literal <root> element in the output. Inside the <root> element <xsl:apply-templates select="module"/> invokes the processor to find matching templates for all <module> elements that are children of the current <fhx> element. Before finding matching templates, the processor is instructed to sort the <module> elements in ascending order by the tag attribute of each <module>. The @ prefix identifies an attribute name instead of an element name.

```
  <xsl:template match="/fhx">
    <root>
      <xsl:apply-templates select="module">
        <xsl:sort select="@tag" data-type="text" order="ascending"/>
```

```
      </xsl:apply-templates>
    </root>
  </xsl:template>
```

The processor executes our next template for each `<module>` element that is a child of the topmost `<fhx>` element. The module matching template performs a shallow copy of the current `<module>` element to the output using the `<xsl:copy>` element. A shallow copy is one that copies a single node without including attributes or descendants. The result is one `<module>` element in the output for each `<module>` element in the input. Inside `<xsl:copy>` are three more XSLT elements. Because these three lie inside the opening and closing `<xsl:copy>` tags, their results will be output inside the `<module>` element created by the `<xsl:copy>` statement. In this way we can create rich hierarchical output from any type of input XML. The next `<xsl:copy-of>` element performs a deep copy which includes a copy of all attributes and descendants (children, grand-children, and so on). The <xsl:copy-of> element includes a select attribute that identifies four elements separated by the pipe (|) operator which translates as a logical OR. The result is a full copy of `<description>`, `<period>`, `<controller>`, or `<type>` elements that are children of the current `<module>` element.

The final two statements of our <module> matching template instruct the processor to find matching templates for each of the attributes (@*) of the current <module> element and to find matching templates for each <attribute> element that is a child of the current <module> element.

```
  <xsl:template match="module">
    <xsl:copy>
      <xsl:copy-of select="description|period|controller|type"/>
      <xsl:apply-templates select="@*"/>
      <xsl:apply-templates select="attribute"/>
    </xsl:copy>
  </xsl:template>
```

Because Access will not import attributes, the next template must turn these attributes into elements as shown here.

Attribute Normal	Element Normal

`<module tag="FIC-101" plant_area="TANKS" category="FLOW" user="johnsev" time=" 1095903496">`	`<module>`
	`  <tag>FIC-101</tag>`
	`  <plant_area>TANKS</plant_area>`
	`  <category>FLOW</category>`
	`  <user>johnsev</user>`
	`  <time>1095903496</time>`
	`</module>`

Notice that this template does not specify each attribute by name but instead processes all <module> attributes irrespective of name or value. Notice how the name() function is used within curly braces {}. The curly braces provide a means for using an expression to dynamically create an attribute value. The expression inside the curly braces is evaluated, converted to a string, and the results assigned to the attribute. The name() fuction returns the name of the current attribute which is used by the <xsl:element> statement to create an element in the output with the name of the current attribute. The result is creating an element for each attribute where the new element name matches the name of the corresponding attribute. The <xsl:value-of> element outputs the value of the current attribute as the text content or value for the newly created element completing the transformation from attributes to elements.

```
  <xsl:template match="module/@*">
    <xsl:element name="{name()}">
```

```
      <xsl:value-of select="."/>
    </xsl:element>
  </xsl:template>
```

The final template creates a single <attribute> element in the output for each <attribute> child of a <module> in the source XML file. This template is a bit more involved so I won't dissect it line-by-line. Instead, I'll describe what is accomplished by this template and let you figure out how it is achieved. This template must include the following functionality:

1. The source XML document separates parameter definition (<attribute>) from its value (<attribute_instance>). This template must normalize this information into a single Parameter element so the resultant Parameter table in Access will include its value as well as its data type, category, position on the control diagram, etc. The template is applied to each <attribute> so a variable is created that points to an <attribute_instance> of the same name within the same <module> from which the value is output.
2. A <ModuleTag> element must be added to each <Parameter> as a foreign key for relating a parameter to its parent <module> in the database.
3. The XML for the value of an <attribute_instance> differs for different data types. For example, an integer type value 25 is stored as <value><cv>25</cv></value> whereas a named set type is stored as <value><string_value>some string</string_value></value>. The transform must return the value no matter what the data type.
4. To support DeltaV named set data types, the output must include an <Enum> element that will contain the name of the named set found in the <attribute_instance>. This element must be included but empty for non named set data types.
5. An attribute may be assigned multiple categories. A good relational design would have each category as a separate record in a related table. However, for simplicity sake, the category will be stored as a single field in the Parameter table as a comma separated list.
6. The name and type attributes must be transformed to elements.
7. The parameter's position on a control drawing defined by elements <x>, <y>, <h>, and <w> must be transformed as children of the output <Parameter>.

Running the Sample

All of the sample files are available for download including the results of the transformation. Download the samples and give them a try. Try importing the original XML file into MS Access both with and without the XSLT transformation and you will see not only the value of this transform but how well it performs.

You are free to try this stylesheet on your own DeltaV configuration – if you have one. If you need help exporting your configuration to an XML file, If you have other XML data that you would like to import into Access, try importing the data without a transform first. The XML structure may be suitable for import without a transform. If not, you have a great way to use what you've learned here and possibly expand your reach with XSLT.

Conclusion

There is much more to learn about the XSLT language and its application however it is a relatively easy language to master compared to a procedural language like VB. With a little practice you'll find you can solve many problems quite quickly that otherwise may have been too time consuming to consider.

If you want to learn more about XSLT, you'll find extensive on-line content and hundreds of informative publications. My personal favorite is XSLT Programmer's Reference by Michael Kay from Wrox Press. The second edition covers XSLT 1.0 and 1.1. The 3rd edition is also available which covers XSLT 2.0.

The next and final installment of this 4 part tutorial will provide more practical examples of using XSLT to transform XML data into something more useful.

1 Sidebars and Graphics

Sidebar – Template Match Pattern Examples

Match Pattern	***Meaning***
`match="/"`	Matches the document root. This is a useful match pattern for generating content at the very top and bottom of the output document. This will only be matched once for the entire document.
`match="Phase"`	This will match any `<Phase>` element.
`match="Phase/Parameter"`	This matches any `<Parameter>` element that is a child of a `<Phase>` element.
`match="Parameter[Type='Real']`	Matches any `<Parameter>` element that contains a `<Type>` element containing the text `Real`. It will look something like this `<Parameter><Type>Real</Type></Parameter>`. Notice that this pattern matches the `<Parameter>` element. The pattern inside the square brackets [] is filter criteria. This pattern will not match `<Parameter>` elements that have no child `<Type>` element and it will not match `<Parameter>` elements with a `<Type>` element that contains the text `Integer`.
`match="Parameter/*"`	Matches any element that is the child of a `<Parameter>` element.
`match="@*"`	Matches any attribute. Notice that an attribute is signified by the `@` character. Therefore the match pattern `Parameter` will match any element of that name whereas the match pattern `@Parameter` will match an attribute of that name.

Sidebar - Frequently Used XSLT Elements

Element	Description
`<xsl:apply-templates>`	Selects a set of nodes from the source XML document and processes each node individually by matching the node with another template in the XSLT document.
`<xsl:attribute>`	Outputs an element attribute. Must be used inside a new element before any other content for the element is output.
`<xsl:call-template>`	Calls a named template. A template may have a name attribute that allows calling the template by name instead of using a match pattern.
`<xsl:choose>`	Provides conditional execution of multiple options similar to a Select Case statement in VB. The options are defined by child elements `<xsl:when>` and `<xsl:otherwise>` which is equivalent to Case Else in VB.
`<xsl:copy>`	Copies the current node from the input XML document to the output. This is a shallow copy that does not copy child nodes or attributes. For a deep copy, see `<xsl:copy-of>`.
`<xsl:copy-of>`	Performs a deep copy of the current node from the source XML document to the output. This copies a node, its attributes, and all descendants.
`<xsl:element>`	Creates a new element in the output. The element name is defined by name attribute. This is an alternative to using a literal element when the element's name is to be determined at runtime.
`<xsl:for-each>`	Selects a set of nodes for processing similar to For-Each-Next in VB. The select attribute is an XPath expression that identifies the nodes for processing.
`<xsl:if>`	Conditional processing that executes when the test attribute expression returns true. Unlike most languages, there is no else or else-if. Use `<xsl:choose>` in those cases.

<xsl:number>	This element can assign a sequential number to the current node and format the number for output.
<xsl:output>	A top level element that controls the output format. This element must appear before any templates.
<xsl:param>	May be used as a top level element for creating global parameters or immediately within an <xsl:template> for creating local parameters. Parameter values may be passed into the stylesheet for global parameters, or into the template for local parameters. A param is analogous to an argument passed to a function in VB.
<xsl:sort>	Provides sorting of nodes selected by <xsl:apply-templates> or <xsl:for-each>.
<xsl:stylesheet>	The outermost element of an XSLT document. The element <xsl:transform> may be used in its place.
<xsl:template>	Defines a template for creating output. It may execute by matching nodes to a pattern or calling the template by name.
<xsl:text>	Will output literal text. Used in favor of literal text when fine control of the output required. Frequently used for transforms that generate simple text output such as a CSV or INI file. This element may not contain any other elements.
<xsl:value-of>	Writes the string value of an expression to the output.
<xsl:variable>	May be used as a top level element for creating global variables or within a template for creating local variables. Unlike <xsl:param> the value cannot be passed in from an external call.

Sidebar – XSLT Processors

There are many XSLT processors available for download such as Saxon, Xalan, and Microsoft MSXML. If you are working on a Windows XP computer, you already have a version of MSXML – most likely version 3 or version 4. All of the samples in this article were tested against MSXML4. The .NET framework also provides a .NET XSLT processor as well. Command line versions are available online for the XSLT processors mentioned here.

Different processors may produce slightly different results or may provide different extension functions that are not portable so to avoid problems it is a good idea to develop your transformations with a specific processor in mind.

Sidebar - XPath

Another standard from the W3C called XML Path Language (XPath) was developed for addressing parts of an XML document from within XSLT. XPath is a sort of query language for returning data subsets from an XML document that has its own syntax and rules like most languages. Within an XSLT stylesheet, an XPath expression may be used anywhere an XSLT element includes a `select=""` attribute.

XPath expression look very similar to the pattern matching expressions in <xsl:template match=""> however the syntax and meaning of these expressions (match expression vs. select expression) is not the same. Most XSLT references provide a comparison of the differences between match expressions and select expressions otherwise known as XSLT Pattern Matching and XPath Expressions.

Sidebar – XSLT Development

A variety of XSLT development tools are available to assist the development of XSLT stylesheets. Most tools include debugging, setting break points, statement completion, help files, XML authoring, and many of the features one expects from a modern development environment. Popular tools are available from Altova and Stylus Studio. These companies provide an extensive library of tools for XML and the technologies designed around XML.

My personal favorite XSLT development tool is Xselerator from Marrowsoft. This application is extremely intuitive and fast. Specifically designed for XSLT development it includes a great set of features without a lot of extra "fluff". Unfortunately this product has not been updated in two years leaving me with the impression that it has a limited future. Still, I have not found a suitable replacement and until then, I'm stickin' with it.

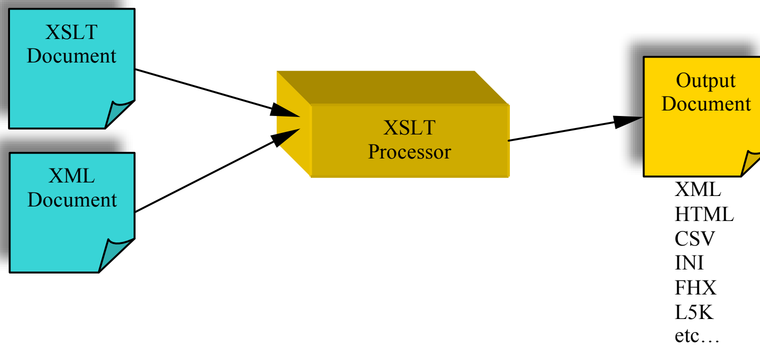
Sidebar – Source XML Before Transform

```
<module tag="AG-100-11" plant_area="T_100" category="" user="JOHNSEV" time="1098221982">
  <description>T-100 Agitator</description>
  <period>1</period>
  <controller>PRODUCTION</controller>
  <primary_control_display>ProdSupport</primary_control_display>
  <instrument_area_display>VMC</instrument_area_display>
  <detail_display></detail_display>
  <type>Control Module</type>
  <sub_type>VMC</sub_type>
……
  <attribute name="AI-IN" type="EXTERNAL_REFERENCE">
    <connection>INPUT</connection>
    <rectangle>
      <x>20</x>
      <y>390</y>
      <h>20</h>
      <w>140</w>
    </rectangle>
    <group>I/O</group>
    <category>
      <category>FIXED_CONNECTOR</category></category>
  </attribute>
  <attribute name="CFM-RUN-IGN" type="ENUMERATION_VALUE">
    <connection>INTERNAL_SOURCE</connection>
    <rectangle>
      <x>280</x>
      <y>1380</y>
      <h>20</h>
      <w>140</w>
    </rectangle>
    <group>Tuning</group>
    <category>
      <category>ADVANCED</category></category>
    <category>
      <category>CFGVIEW</category></category>
  </attribute>
  <attribute name="HOLD-RESET" type="BOOLEAN">
    <connection>INTERNAL_SOURCE</connection>
    <rectangle>
      <x>20</x>
      <y>1020</y>
      <h>20</h>
      <w>140</w>
    </rectangle>
    <group>Operating</group>
    <category>
      <category>COMMON</category></category>
  </attribute>
  <attribute_instance name="AI-IN">
    <value>
      <ref>//SIC-100-11/AO</ref>
    </value>
  </attribute_instance>
  <attribute_instance name="CFM-RUN-IGN">
    <value>
      <set>YES_NO</set>
      <string_value>No</string_value>
      <changeable>F</changeable>
    </value>
  </attribute_instance>
  <attribute_instance name="HOLD-RESET">
    <value>
      <cv>F</cv>
    </value>
  </attribute_instance>
</module>
```

Sidebar – XML After Transform – Ready for MS Access Import

```
<module>
  <tag>100-AGIT</tag>
  <description>T-100 Production Agitator Control</description>
  <period>1</period>
  <controller>PRODUCTION</controller>
  <type>EM</type>
  <plant_area>T_100</plant_area>
  <category></category>
  <user>JOHNSEV</user>
  <time>1095903496</time>
  <Parameter>
     <ModuleTag>AG-100-11</ModuleTag>
     <name>AI-IN</name>
     <type>EXTERNAL_REFERENCE</type>
     <connection>INPUT</connection>
     <group>I/O</group>
     <x>20</x>
     <y>390</y>
     <h>20</h>
     <w>140</w>
     <categories>FIXED_CONNECTOR</categories>
     <enum></enum>
     <value>//SIC-100-11/AO</value>
  </Parameter>
  <Parameter>
     <ModuleTag>AG-100-11</ModuleTag>
     <name>CFM-RUN-IGN</name>
     <type>ENUMERATION_VALUE</type>
     <connection>INTERNAL_SOURCE</connection>
     <group>Tuning</group>
     <x>280</x>
     <y>1380</y>
     <h>20</h>
     <w>140</w>
     <categories>ADVANCED,CFGVIEW</categories>
     <enum>YES_NO</enum>
     <value>No</value>
  </Parameter>
  <Parameter>
     <ModuleTag>AG-100-11</ModuleTag>
     <name>HOLD-RESET</name>
     <type>BOOLEAN</type>
     <connection>INTERNAL_SOURCE</connection>
     <group>Operating</group>
     <x>20</x>
     <y>1020</y>
     <h>20</h>
     <w>140</w>
     <categories>COMMON</categories>
     <enum></enum>
     <value>F</value>
  </Parameter>
</module>
```

Graphic of XSLT Transformation Process



Cascade Solutions
SUCCESS DELIVERED

18312 West Creek Drive
Tinley Park, IL 60477
P: 708.802.6000 F: 708.802.6008
www.cascadesolutions.com

The ABCs of XML
Part 4 – Putting It All Together

By John T. Sever
Cascade Controls, Inc.

Introduction

In the final installment of this series, I'll provide practical XSLT samples for transforming XML. Also, I'll identify a few common problems you are likely to encounter.

Problem 1 – Why do I get extra text in my output? Answer: Built-in Templates

XSLT beginners often find unwanted unstructured text outside of carefully designed and highly structured output. The extra text is likely the result of the XSLT processor applying a *built-in template*. When `<xsl:apply-templates>` is invoked and no matching template is found in the stylesheet, the processor invokes a default built-in template that lives inside the processor.

One built-in template matches the document root or any element node then calls `<xsl:apply-templates>` to process the children of the current node. This template gives XSLT its automatic recursive feature for all elements that have no explicit template in your transform.

The built-in template for text nodes (text between element open and close tags) copies the text to the output. This built-in template causes great confusion if you don't understand the way the processor handles unmatched nodes. You can easily see the results of this behavior by transforming an XML document with a stylesheet that contains no templates as shown here.

```
<xsl:stylesheet version="1.0"  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
</xsl:stylesheet>
```

There are also built-in templates for comments and processing instructions that do absolutely nothing.

Built-in templates have a lower priority than all other templates. Thus, you may override a built-in template by creating your own template.

To override the built-in template for text nodes, simply include the following template in your stylesheet.

```
<xsl:template match="text()"/>
```

If you have other templates for processing text nodes, their match patters will be more specific than this pattern and will therefore be matched by the XSLT processor as having greater priority over this template. Add this one line template to a blank stylesheet and see that it produces a blank output file – a great starting point when developing new transforms.

Problem 2 – None of my templates work when the XML file includes a default namespace.

Dealing with a default namespace can be a little tricky. Because the designers of XSLT decided that terseness is of minimal importance, XSLT is a language that is verbose and explicit in most cases. Default namespaces are uncharacteristically non-explicit and can result in confusion and frustration. Remember that all elements in an XML document must belong to a namespace. In fact there is no way to create an XML element that is not part of any namespace. This may seem confusing if you often work with XML documents that include no namespace declarations. Document elements with no declared namespace actually belong to a *default null namespace* and require no special namespace modifier in match patterns or XPath expressions. A non-null *default namespace* is one that does not include a namespace alias prefix. The following examples will illustrate this fine point.

In this sample, `RecipeElement` and `OtherElements` are included in the default null namespace and may be matched in XSLT expressions as `RecipeElement` or `OtherElements` respectively.

```
<RecipeElement>
  <OtherElements/>
</RecipeElement>
```

In this sample, `RecipeElement` and its children (`OtherElements`) belong to the default namespace `urn:Rockwell/MasterRecipe` because the namespace does not include a prefix after `xmlns`. These will no longer be matched by `RecipeElement` and `OtherElements`. To match these requires the namespace be declared with a prefix in your stylesheet for example `xmlns:rs="urn:Rockwell/MasterRecipe"`. Remember the value of this attribute must exactly match (case sensitive) the namespace as it is declared in the XML file. This declaration includes the prefix rs that may now be used for match expressions such as `rs:RecipeElement` and `rs:OtherElements` respectively. Remember that because the namespace is declared as a default namespace in the XML document, all child elements inherit from this same namespace unless explicitly overridden by use of another namespace prefix.

```
<RecipeElement xmlns="urn:Rockwell/MasterRecipe">
  <OtherElements/>
</RecipeElement>
```

In this sample, the namespace is assigned the explicit prefix `rs` and therefore is no longer the default namespace. Because no default namespace is declared and because `RecipeElement` does not use the `rs` prefix, it belongs to the default null namespace and may be matched as `RecipeElement` and `OtherElements` respectively.

```
<RecipeElement xmlns:rs="urn:Rockwell/MasterRecipe">
  <OtherElements/>
</RecipeElement>
```

In this sample, the namespace is assigned a prefix which is **not** used by `RecipeElement`. Therefore, matching this element requires a namespace declaration with a prefix in your stylesheet. Although it may be confusing, the XSLT stylesheet prefix is not required to match the XML source document prefix because a prefix is only a shorthand local alias of the full namespace. However, I recommend using the identical prefixes in your transforms to avoid confusion.

You may be surprised to learn that `OtherElements` is included in the default null namespace! Only default namespaces are inherited by child nodes. The prefixed namespace used in this sample is not the default namespace and therefore is not inherited by `OtherElements`. This means that `RecipeElement` and `OtherElements` are members of different namespaces.

```
<rs:RecipeElement xmlns:rs="urn:Rockwell/MasterRecipe">
  <OtherElements/>
</rs:RecipeElement>
```

I used the top of an RSBatch master recipe for the above samples. Rockwell uses this default namespace for each recipe that is saved in XML format. Therefore, your transformation for any RSBatch recipe should look like this.

```
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                              xmlns:rs=" urn:Rockwell/MasterRecipe"
                              exclude-result-prefixes="rs">

  <xsl:template match="rs:RecipeElement">
     …
  </xsl:template>
</xsl:stylesheet>
```

Understanding the use of namespaces in XML and XSLT can be one of the most frustrating aspects of developing XSLT transforms if you don't understand these fine points so I suggest you re-read this section and make sure you understand how default namespaces differ from prefixed namespaces. A little experimentation can help too.

Problem 3 – My stylesheet automatically adds namespace attributes to output elements. Can I remove these?

You can never produce an output file using namespace prefixes that have not been declared and therefore each namespace used in the output must be declared at least once. However, you may find that your output is littered with namespace declarations in nearly every element whether it is used by that element or not. To suppress unnecessary namespace declarations, use the attribute `exclude-results-prefixes` in the `xsl:stylesheet` element (see previous example). The value of this attribute is a list of namespace prefixes separated by white space. Remember this will not remove all namespace declarations but it will remove unused and unnecessary namespace declarations.

Problem 4 – Text Output

Generating text output such as a CSV file (comma separated values) can prove to be more difficult than you may expect until you understand how the processor handles text and white space. Often you may find it difficult to generate one particular character like a quotation mark (") because it has specific meaning in XML or by the XSLT language. Here are a few pointers for generating plain text output.

1. Make sure to include the output element before your templates as follows: <xsl:output method="text"/>
2. Use concat() function when creating output that is a combination of many text nodes or a mix of text nodes and literal text. This sample will generate a single CSV row with two comma separated columns where each value is contained within quotation marks.
   ```
   <xsl:value-of select="concat('&quot;',Value1, '&quot;,&quot;', Value2, '&quot;&#10;')"/>
   ```
3. Use the <xsl:text> element over literal text in your templates. This will give you more explicit control of your output while maintaining an easy to read transform. XSLT elements are not allowed inside an <xsl:text> element. The following verbose sample will produce the same output as the previous sample.
   ```
   <xsl:text>"</xsl:text><xsl:value-of select="Value1"/><xsl:text>","</xsl:text><xsl:value-of
   select="Value2"/><xsl:text>"&#13;</xsl:text>
   ```

4. Output special characters using character entity escape sequences. The most commonly used character entities are shown here:

Character	***Sequence***
"	"
TAB	
New Line
&	&
<	<
>	>
'	'

5. Use a CDATA section to simplify using special characters because everything inside of CDATA is ignored by the XML parser. Again, this will not work if your output is to be a mixture of literal text and XSLT elements like the previous example. A CDATA section begins with **<![CDATA[** and ends with **]]>**.

The following examples demonstrate practical ways that XSLT can be used in your daily work. XML, XSLT, and supporting information for each example is available for download at www.????

Example 1: Import DeltaV Modules into MS Access Database

This example fills out the transform that I created for Part 3 of this series.

Example 2: Convert RSBatch Recipe to a Word Document

This example transforms a Rockwell RSBatch recipe XML file into an MSWord 2003 document. MSWord 2003 supports an XML structure (schema) that includes every feature available from inside Word 2003. A variety of information regarding XML in MS Office 2003 is available at http://msdn2.microsoft.com/en-us/office/aa905548.aspx. From the References section on this page, you may download and install the Office 2003 XML Reference Schemas. This download includes a help file describing the proper structure for creating a Word document with XML.

If it has not yet sunk it, you can create a Word document with a text editor and some well formed XML – without installing MS Word! This means that our transform must simply convert the Rockwell RSBatch XML structure into a Word 2003 structure that can then be opened, viewed, printed, or edited inside MS Word 2003.

If you spend some time looking over the schema for MS Word 2003 you will notice that it is extremely verbose. Most of our engineers can use XSLT effectively but I've found that learning WordProcessingML (XML for MS Word 2003) is too much to ask of my control engineers. To bridge the gap, we have created our own simplified intermediate XML language that we call CascadeDocML. This simplified schema is much easier for our engineers to learn as it supports the features we use most in Word. To support CascadeDocML, we created an XSLT transform that converts from our simplified syntax to full WordProcessingML syntax. Therefore, to generate a Word document is a two step process from Input XML, to CascadeDocML, to WordProcessingML.

This is the methodology used for this example. The download for this example includes a document describing the functionality and structure of CascadeDocML so you may use it for your own

applications. It supports a small subset of WordProcessingML but it is significantly easier to learn than WordProcessingML.

Example 3: Search and Replace

This example is used to generate an output file that is an exact copy of the input XML file with keywords replaced according to a list of search replace pairs defined in a separate file. This transform may be used to duplicate a complete unit configuration where only the tags have changed. This transform is different than performing search/replace in a text editor like Notepad as it does not replace everything throughout the file. Our sample input file is a DeltaV XML file in which we want to be very selective about what code elements are available for search/replace. For example, if you want to replace 100 with 200 in Notepad, you may end up modifying a setpoint value or drawing x/y coordinates instead of just modifying module tags.

Although this example may not work perfectly for your specific application, it is extremely modular and easy to extend to your own needs.

Example 4: Generate Parameter Spreadsheet from RSBatch Area Model

This sample will create an Excel spreadsheet of an RSBatch Area Model with each process cell, unit, and phase. Also each phase will include a detailed list of the phase parameters and reports. There are two transforms for this sample. One transform creates a CSV text file that may be opened in Excel. The other creates an Excel XML file instead of a CSV. The Excel XML output includes spreadsheet formatting that is much more complete than a simple CSV file.