RC23144 (W0403-066) March 9, 2004
Computer Science

IBM Research Report

Using XSLT to Detect Cycles in a Directed Graph

David Marston
IBM Research Division
Thomas J. Watson Research Center
One Rogers Street
Cambridge, MA 02142

Research Division
Almaden - Austin - Beijing - Haifa - India - T. J. Watson - Tokyo - Zurich

Using XSLT to Detect Cycles in a Directed Graph

David Marston
IBM Research Division
Thomas J. Watson Research Center
One Rogers Street
Cambridge, MA 02142, USA

Abstract

This paper describes the most difficult stage of a multi-stage process in which a model of a business process is converted from a legacy system to a format digestible by IBM's WebSphere Business Integration Modeler. The stage where cycles must be detected in a directed graph was initially suspected to be beyond the abilities of generic XSLT, which was the tool of choice for the preceding and following stages. The paper describes a well-known algorithm for detecting cycles and then discusses how it can be implemented in XSLT, followed by some observations about the implementation.

Introduction

WebSphere Business Integration (WBI) Modeler, IBM's product [1] for analyzing business process models, can import a model expressed in XML. The model is a directed graph of objects (tasks, decision branches, etc.) and edges ("connectors" between objects). A loosely-assembled team of people at IBM Research has been exploring ways to manipulate representations of processes to make them suitable for importation. When the model is represented in XML, one can use XSLT [2] to alter the structure to meet the needs of WBI Modeler. Prior to the activity reported on here, we had addressed several other needs for XML restructuring on our prototype. The final issue was that WBI Modeler requires that the directed graph contain no cycles at the time the model is to be imported.

Since we had used XSLT to produce all the required elements and attributes, and to set them in the correct nesting and order, it was desirable to use XSLT for breaking cycles, so that only one tool and method would suffice for all manipulation of XML data. The prominent sources for non-WBI models have the ability to export XML, in which case XSLT could be used for the entire process of transforming the model from one environment to the other. There may be more than one XSLT transformation stage in the pipeline, which gives the business process analyst the opportunity to tweak the intermediate XML for a more faithful representation of the process.

I will now focus on the transformation that detects back-links that form cycles in the graph and earmarks them for special treatment. I assume that earlier stages of the overall transformation process reduced the source data to just those XML nodes (elements, attributes, text nodes) needed by WBI Modeler, plus some attributes containing identifiers. I assume that later stages will remove the identifiers that WBI Modeler won't recognize, and will generally clean up all the data as required. The model that WBI Modeler will import will have <Connector> elements for each connector-like structure in the original model, except for those that were determined to be back-links. The replacement for each back-link could be two new <ExternalProcessObject> elements and a new <Connector> element from each, representing two stub ends, if the business process analyst wants to have a visible record of where the links were broken.

Cycle Detection Method

The algorithm described here performs a depth-first traversal of the graph. It must eventually analyze every edge (<Connector> element) linking two nodes. In deference to business process analysts, I will henceforth use the interchangeable terms "link" and "connector" rather than "edge." When representing a business process as a graph, there will be at least one starting node, which is a node that has no inbound links. Such a node can be deemed to represent the process being ready for initiation, to be triggered by a request for the service that the business process provides.

When the transformation starts, every node of the graph is unvisited and no connectors have been analyzed. Starting with one of the designated starting nodes, we change the state of the node from Unvisited to Pending, and select an outbound link for exploration. Following the link will take us to another node that had been Unvisited, now becomes Pending, and the link itself can be designated as "Used" and normal. Eventually, we expect the depth-first search will arrive

at a node that has no outbound links (an ending node in the process), in which case we change the state of that node from Unvisited to Finished, and back up to the prior node, which is in the Pending state. When all outbound links from a node have been Used, we change the node from Pending to Finished and back up to the prior node. It is possible to select an outbound link and discover that the node at the other end is already in the Finished state, in which case we need only mark the link as Used. Finally, it is possible to select an outbound link and discover that the node at the other end is in the Pending state, in which case the link is one of the back-links that must be flagged. Colloquially, while exploring all the nodes beyond a Pending node, we found a link pointing back to the same node, creating an infinite loop.

The WBI-compatible XML representation of the graph has a flat list of Connectors and flat lists for each type of node. This implies that the action of "moving" forward or backward over a link devolves to changing the identifier of the current node. One way to move backward is to pop a node identifier off a stack, without reference to the Connector between them. This fact is useful in a graph with more than one starting node, because it simplifies the process of changing to a new starting node after one starting node has been sent to the Finished state. We simply pre-load the stack with a list of all starting nodes, and continue processing until the stack is exhausted.

XSLT Implementation of the Algorithm

The XSLT stylesheet that implements this stage of the process is given in the Appendix. In summary, it copies through all objects unchanged except for <Connector> elements that are back-links, which will get a special attribute (BackLink="yes"). The back-link connectors are otherwise retained, to be broken at a later stage. This stage does not have to be isolated, but there are advantages to doing so, as described in the next section. The lists of nodes in Pending and Finished states and Connectors in Used or BackLink states are maintained in ordinary XSLT variables, so the stylesheet uses only standard XSLT capabilities and thus can be executed by any conformant XSLT processor.

The need to maintain lists of nodes and Connectors having certain states requires that an identifier scheme be present. The stylesheet given here uses an attribute named "ni" for every node (regardless of node type) and an attribute named "ci" for every Connector. Every <Connector> has sub-elements <Source> and <Target> to express the directionality of the link, and we require each to have an attribute named "li" that references the "ni" attribute of the associated node. By requiring each to be a valid XML name, we ensure that the values of the ci/li/ni attributes will never contain a plus (+) character, freeing + to be used safely as a delimiter. The names must be chosen so that one name will never be a sub-string of another name. This scheme of identifiers will meet our needs and can be, but does not need to be, independent of any other identifiers found in the source. If the source model (as expressed in XML) does not already have unique identifiers that are compact and meet the naming constraints, an earlier stage should produce such identifiers. WBI Modeler cannot accept any unknown attributes, so the identifiers must be removed when producing the XML that is the result of the final-stage transformation.

The implementation of the lists of nodes and connectors is adapted to the XSLT/XPath 1.0 function set. The stack is the same as the other lists, except that position matters. Each list is a string of identifiers separated by plus signs. A new item is added to the list via the string concatenation function (concat). The presence of an item on the list is tested by the "contains" function, which returns a boolean. The Finished and BackList lists (of nodes) and the Used list (of Connectors) only grow, so no other list management tools are needed for those. Items are removed from lists by use of "substring-before" and "substring-after" and concatenation if necessary. The Pending and Stack variables are lists of nodes that grow and shrink. For the Stack variable, node identifiers (and plus-sign separators) are added to and removed from the front of the string, which is optimal for the "substring-after" function.

Variables in XSLT cannot be altered once set. When XSLT processing must retain information for later use, recursion of xsl:call-template is usually necessary. The first use of such recursion in the stylesheet is to take an XML node-set of all the starting nodes and pass it to the MakeStart template, intending to convert the node-set to a stack in our list notation. Each call to MakeStart takes one node from the node-set and adds it to the stack, then calls MakeStart recursively with a decremented index number and the latest version of the list being assembled. When the index reaches 1, the template emits the entire list (via xsl:value-of) and stops recursing. Later uses of recursion are more elaborate in detail but the same in principle.

In XSLT terms, the main step-wise process of traversing the graph always goes "forward" in the sense that the recursion gets deeper with every step, regardless of whether we are getting deeper in the graph. We step to a new node by calling the Step template with the Node parameter set to the node identifier of the destination node, regardless of whether that node is deeper, shallower (i.e., backing up), or unrelated (i.e., going to the next starting node). When we need to add a Connector to the Used list of connectors, we also call Step recursively, but don't change the Node parameter. Every recursive call to Step changes at least one of the six parameters.

When the Step template is called, it has to choose among five cases, which are ordered for efficient operation. The first decision point is to determine whether the current node has any outbound Connectors. If none, then we can add the current node to the Finished list and then pop the Stack list to obtain the "next" node, which is the one we just left to get to the current node. If there are outbound Connectors, we try to find an unused one. If all outbound Connectors are on the Used list, we add the current node to the Finished list and back up, as in the previous case. Otherwise, there is at least one Connector that is currently unused, and we now must pick one and look at the state of the Target node to which the Connector leads. Determination of the unused outbound Connector or lack thereof is accomplished by calling the FindUnused template, which either returns a Connector identifier or a non-conflicting string ("!NONE!" in the prototype) that signals that no more are unused. If the Target is on the Finished list, we simply mark the Connector as Used, stay put on the current node, and look for another unused outbound Connector. If the Target is on the Pending list, we do the same as with a Finished node and also add the Connector identifier to the list of back-links. If the Target is on neither (Finished nor Pending) list, then it is unvisited and we can make a conventional forward step onto the node. All changing of lists and the Node value is done at the time parameters are set for the next call to Step, which is a standard XSLT technique. The exit point of the recursion is when we have just added a node to the Finished list, attempt to get the next node identifier from the Stack list, and find nothing there, which can occur in either of the first two cases enumerated above. As we exit, we emit the list of back-links ($BackList), which was the objective of the recursion.

The FindUnused template also operates recursively. It steps through a set of Connectors in reverse document order, returning the identifier of the first unused Connector encountered (last in document order), or "!NONE!" if the list is exhausted. The use of reverse document order means that it is possible to get different results by changing the order of <Connector> elements in the input. The utility of such changes is discussed below.

Tuning for Better Results

A back-link is one of at least two links leading into a node that also has an outbound link. A person who studies the process may observe that a particular inbound link is appropriate to be designated as the back-link, yet the process may choose another link. For example, one graph had a link labeled "Retry" that clearly represented a cycle in the process, and I wanted that one to be designated as a back-link. The first inbound link encountered won't be flagged as a back-link because the target node was in the Unvisited state at the time. If the back-link flag is not being applied to the best choice, the <Connector> elements can be rearranged in the input document. Recall that all <Connector> elements appear in a flat list, and that FindUnused searches a flat sub-list of the <Connector> elements, those whose Source is the current node, working in

reverse document order. Selecting a <Connector> that is appropriate to be flagged as a back-link and moving it up toward the beginning of the XML document will increase the likelihood that it will be tested later and thus be identified as a back-link. Only one move should be made at a time, because the effects can spread through large portions of the graph. In trials with the prototype, I could cause the number of back-links to vary from 3 to 6 in a graph that had 4 links deserving to be characterized as back-links.

One could use XSLT extensibility features to provide an alternate way to manage the lists of nodes and connectors. If the cost of string matching can be eliminated or reduced, then performance gains might be noticeable. The graph must still be traversed in depth-first order, so an XSLT implementation will still need to operate recursively.

In all five cases within the Step template, the recursive call to Step is the last operation in its branch of the program. This allows tail-recursion elimination to be applied. In my prototype work, I used a Java-based XSLT processor (Xalan/Java) and a Java "JIT" facility [3] that performs the tail-recursion elimination. The optimized version runs in a reasonable amount of time and memory, admitting the possibility of expanding the stylesheet given here to encompass more stages of the transformation work. Having a break point in the pipeline after the stage where back-links are flagged allows manual review of the substantive transformations. Later stages merely finalize the data for importation.

Conclusion

I have demonstrated that XSLT can be used to find and break cycles in a directed graph, when that graph is expressed in XML with a separate element for each connector. In conjunction with other XSLT transformations, a business model in XML can be presented in the form that WBI Modeler requires for its XML import facility.

References

[1] http://www.ibm.com/software/integration/wbimodeler
[2] XSL Transformations (XSLT) 1.0; http://www.w3.org/TR/1999/REC-xslt-19991116
[3] T. Suganuma et al., "Overview of the IBM Java Just-in-Time Compiler", IBM Systems Journal, Vol. 39, No. 1

Appendix

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

  <!-- FileName: CycleFind.xsl -->
  <!-- Creator: David Marston -->
  <!-- Purpose: Find cycles by moving nodes from Unvisited to Pending to
Finished.
    Input must have node identifier (@ni) and connector identifier (@ci)
attributes. -->

<xsl:output method="xml" encoding="UTF-8" indent="yes"/>

<xsl:template match="WFBPR"><!-- the document element -->
  <WFBPR>
    <xsl:copy-of select="RepositoryData"/>
    <xsl:apply-templates select="Processes"/>
  </WFBPR>
</xsl:template>
```

```
<xsl:template match="Processes">
  <Processes>
    <xsl:apply-templates select="Process"/>
  </Processes>
</xsl:template>

<xsl:template match="Process">
  <!-- simple copying continues until we reach the Connectors element
-->
  <Process>
    <xsl:copy-of select="ProcessName"/>
    <xsl:copy-of select="ProcessType"/>
    <xsl:copy-of select="TaskObjects"/>
    <xsl:copy-of select="PhiObjects"/>
    <xsl:copy-of select="ExternalProcessObjects"/>
    <xsl:copy-of select="DecisionObjects"/>
    <Connectors>
      <!-- Now walk through the graph. Start by finding all the starting
nodes. -->
      <xsl:variable name="Starts"
select="ExternalProcessObjects/ExternalProcessObject[@Start]/@ni"/>
      <xsl:if test="count($Starts) = 0">
        <xsl:message terminate="yes">
          <xsl:text>No starting nodes found!</xsl:text>
        </xsl:message>
      </xsl:if>
      <!-- Convert the node-set ($Starts) to a list, which will begin
our stack. -->
      <xsl:variable name="StartList">
        <xsl:call-template name="MakeStart">
          <xsl:with-param name="list" select="$Starts"/>
          <xsl:with-param name="lSeq" select="count($Starts)"/>
          <xsl:with-param name="work" select="'+'"/><!-- empty list -->
        </xsl:call-template>
      </xsl:variable>
      <!-- Now pop off the first member of the start list as the
starting node. The rest go into $Stack. -->
      <xsl:variable name="begin"
select="substring-before($StartList,'+')"/>
      <!-- The goal of the recursive stepping is to populate the list in
$BackLInks. -->
      <xsl:variable name="BackLinks">
        <!-- Now make the initial call to Step. -->
        <xsl:call-template name="Step">
          <xsl:with-param name="Node" select="$begin"/>
          <xsl:with-param name="Pending" select="$begin"/><!-- this node
will be in Pending state when we go in -->
          <xsl:with-param name="Finished" select="''"/><!-- empty list
-->
          <xsl:with-param name="UsedConn" select="''"/><!-- empty list
-->
          <xsl:with-param name="BackList" select="''"/><!-- empty list
-->
          <xsl:with-param name="Stack"
select="substring-after($StartList,'+')"/>
        </xsl:call-template>
      </xsl:variable>
      <!-- At this point, we have detected all back-links and put them
on the $BackLInks list. -->
```

```
      <!-- Copy each Connector straight through unless flagged as a
back-link. -->
      <xsl:for-each select="Connectors/Connector">
        <xsl:choose>
          <xsl:when test="contains($BackLinks,@ci)">
            <!-- This Connector is a back-link. Put a special attribute
to mark that fact. -->
            <Connector BackLink="yes">
              <!-- then copy all other attributes and descendants -->
              <xsl:copy-of select="./@*"/>
              <xsl:copy-of select="./node()"/>
            </Connector>
          </xsl:when>
          <xsl:otherwise>
            <!-- It's a normal connector, so just copy it through. -->
            <xsl:copy-of select="."/>
          </xsl:otherwise>
        </xsl:choose>
      </xsl:for-each>
    </Connectors>
  </Process>
</xsl:template>

<!-- Named templates below, alphabetcally by name. -->

<xsl:template name="FindUnused">
  <!-- Look for unused connector in $conns list. Return ci of unused or
"!NONE!". -->
  <xsl:param name="conns"/>
  <xsl:param name="fSeq"/>
  <xsl:param name="used"/><!-- A copy of the latest list of all
connectors used so far. -->
  <xsl:choose>
    <xsl:when test="contains($used,$conns[$fSeq]/@ci)">
      <!-- This one is already used. Are there more? -->
      <xsl:choose>
        <xsl:when test="$fSeq=1">
          <xsl:text>!NONE!</xsl:text>
        </xsl:when>
        <xsl:otherwise>
          <!-- There are more, so recurse with a lower $fseq number. -->
          <xsl:call-template name="FindUnused">
            <xsl:with-param name="conns" select="$conns"/>
            <xsl:with-param name="fSeq" select="$fSeq - 1"/>
            <xsl:with-param name="used" select="$used"/>
          </xsl:call-template>
        </xsl:otherwise>
      </xsl:choose>
    </xsl:when>
    <xsl:otherwise><!-- "Return" the @ci of this connector as the next
unused one. -->
      <xsl:value-of select="$conns[$fSeq]/@ci"/>
    </xsl:otherwise>
  </xsl:choose>
</xsl:template>

<xsl:template name="MakeStart">
  <!-- Take members of node-set $list and form a delimited concatenation
of their string values. -->
  <xsl:param name="list"/>
```

```
  <xsl:param name="lSeq"/>
  <xsl:param name="work"/>
  <xsl:choose>
    <xsl:when test="$lSeq=1">
      <!-- Down to the last one. Finish off the string and "return" it.
-->
      <xsl:value-of select="concat(string($list[1]),$work)"/>
    </xsl:when>
    <xsl:otherwise>
      <!-- Add the current member (and a delimiter) to the string, then
recurse. -->
      <xsl:call-template name="MakeStart">
        <xsl:with-param name="list" select="$list"/><!-- Don't bother
shrinking this. -->
        <xsl:with-param name="lSeq" select="$lSeq - 1"/>
        <xsl:with-param name="work"
select="concat('+',string($list[position()=$lSeq]),$work)"/>
      </xsl:call-template>
    </xsl:otherwise>
  </xsl:choose>
</xsl:template>

<xsl:template name="Step">
  <!-- Step from one Node to another, or to the same Node with updated
data. This is a 5-branch decision tree. -->
  <xsl:param name="Node"/><!-- ni of the Node we are stepping to -->
  <xsl:param name="Pending"/><!-- string containing ni of all Nodes
(including current $Node) visited but not done -->
  <xsl:param name="Finished"/><!-- string containing ni of all Nodes
visited and resolved -->
  <xsl:param name="UsedConn"/><!-- string containing ci of all
connectors that have been analyzed -->
  <xsl:param name="BackList"/><!-- string containing ci of all
connectors identified as back-links so far -->
  <xsl:param name="Stack"/><!-- string containing ni of Nodes on the
path to here (excluding $Node) -->
  <xsl:message>
    <xsl:text>-----STEP into box </xsl:text>
    <xsl:value-of select="$Node"/>
  </xsl:message>
  <!-- List and count all outbound links from here. -->
  <xsl:variable name="ConnList"
select="Connectors/Connector[Source/@li=$Node]"/>
  <xsl:variable name="TotalC" select="count($ConnList)"/>
  <xsl:choose>
    <xsl:when test="$TotalC = 0">
      <!-- No outbound links, so this ends a path. -->
      <xsl:choose>
        <xsl:when test="string-length($Stack) &gt; 0">
          <!-- More work to do. -->
          <xsl:call-template name="Step">
            <xsl:with-param name="Node"
select="substring-before($Stack,'+')"/><!-- Get "next" node off the
stack. -->
            <xsl:with-param name="Pending"
select="concat(substring-before($Pending,$Node),
              substring-after($Pending,concat($Node,'+')))"/><!--
Current node no longer pending. -->
```

```
            <xsl:with-param name="Finished"
select="concat($Node,'+',$Finished)"/><!-- Add current node to Finished
list. -->
            <xsl:with-param name="UsedConn" select="$UsedConn"/><!-- We
are backing up; no additional usage. -->
            <xsl:with-param name="BackList" select="$BackList"/>
            <xsl:with-param name="Stack"
select="substring-after($Stack,'+')"/><!-- This is the rest of the
stack. -->
          </xsl:call-template>
        </xsl:when>
        <xsl:otherwise>
          <!-- The stack is empty; "Return" the list of back-links. -->
          <xsl:value-of select="$BackList"/>
        </xsl:otherwise>
      </xsl:choose>
    </xsl:when>
    <xsl:otherwise>
      <!-- This node has some outbound links. Look for unused ones. -->
      <xsl:variable name="NextUnused">
        <xsl:call-template name="FindUnused">
          <xsl:with-param name="conns" select="$ConnList"/>
          <xsl:with-param name="fSeq" select="$TotalC"/>
          <xsl:with-param name="used" select="$UsedConn"/><!-- Must give
it the current snapshot of what's used. -->
        </xsl:call-template>
      </xsl:variable>
      <xsl:choose>
        <xsl:when test="$NextUnused = '!NONE!'">
          <!-- All outbound connectors have been traversed. -->
          <xsl:choose>
            <xsl:when test="string-length($Stack) &gt; 0">
              <!-- More work to do. -->
              <xsl:call-template name="Step">
                <xsl:with-param name="Node"
select="substring-before($Stack,'+')"/><!-- Get "next" node off the
stack. -->
                <xsl:with-param name="Pending"
select="concat(substring-before($Pending,$Node),
                  substring-after($Pending,concat($Node,'+')))"/><!--
Current node no longer pending. -->
                <xsl:with-param name="Finished"
select="concat($Node,'+',$Finished)"/><!-- Add current node to Finished
list. -->
                <xsl:with-param name="UsedConn" select="$UsedConn"/><!--
We are backing up; no additional usage. -->
                <xsl:with-param name="BackList" select="$BackList"/>
                <xsl:with-param name="Stack"
select="substring-after($Stack,'+')"/><!-- This is the rest of the
stack. -->
              </xsl:call-template>
            </xsl:when>
            <xsl:otherwise>
              <!-- The stack is empty; "Return" the list of back-links.
-->
              <xsl:value-of select="$BackList"/>
            </xsl:otherwise>
          </xsl:choose>
        </xsl:when>
        <xsl:otherwise>
```

```
          <!-- We have an unused connector to follow. There are three
possibilities:
            A) Connects to a Node that has been finished. No problem
here.
            B) Connects to a Node whose descendants are still being
explored. This is a back-link.
            C) Connects to a Node that has not been visited before. Step
into it and keep looking. -->
          <xsl:variable name="Follow"
select="Connectors/Connector[@ci=$NextUnused]/Target/TargetName/@li"/>
          <!-- $Follow is the identifier (ni) of the node on the other
end of connector $NextUnused. -->
          <xsl:choose>
            <xsl:when test="contains($Finished,$Follow)"><!-- Case A -->
              <!-- Now stay on $Node, but mark current connector used,
which requires a Step call. -->
              <xsl:call-template name="Step">
                <xsl:with-param name="Node" select="$Node"/>
                <xsl:with-param name="Pending" select="$Pending"/>
                <xsl:with-param name="Finished" select="$Finished"/>
                <xsl:with-param name="UsedConn"
select="concat($NextUnused,'+',$UsedConn)"/><!-- Add to list. -->
                <xsl:with-param name="BackList" select="$BackList"/>
                <xsl:with-param name="Stack" select="$Stack"/>
              </xsl:call-template>
            </xsl:when>
            <xsl:when test="contains($Pending,$Follow)"><!-- Case B:
found a back-link -->
              <!-- Now stay on $Node, but mark current connector used,
which requires a Step call. -->
              <xsl:call-template name="Step">
                <xsl:with-param name="Node" select="$Node"/>
                <xsl:with-param name="Pending" select="$Pending"/>
                <xsl:with-param name="Finished" select="$Finished"/>
                <xsl:with-param name="UsedConn"
select="concat($NextUnused,'+',$UsedConn)"/><!-- Add to list. -->
                <xsl:with-param name="BackList"
select="concat($NextUnused,'+',$BackList)"/><!-- Also add to this list.
-->
                <xsl:with-param name="Stack" select="$Stack"/>
              </xsl:call-template>
            </xsl:when>
            <xsl:otherwise><!-- Case C -->
              <!-- Follow the connector. This is the conventional
forward step. -->
              <xsl:call-template name="Step">
                <xsl:with-param name="Node" select="$Follow"/>
                <xsl:with-param name="Pending"
select="concat($Follow,'+',$Pending)"/>
                  <!-- This is our first visit to the node. Update
$Pending now. -->
                <xsl:with-param name="Finished" select="$Finished"/>
                <xsl:with-param name="UsedConn"
select="concat($NextUnused,'+',$UsedConn)"/>
                  <!-- Update $UsedConn also, because we are using a
previously-unused connector. -->
                <xsl:with-param name="BackList" select="$BackList"/>
                <xsl:with-param name="Stack"
select="concat($Node,'+',$Stack)"/>
```

```
                  <!-- Put the Node we're leaving on the stack, because
we expect to back up to it later. -->
              </xsl:call-template>
            </xsl:otherwise>
          </xsl:choose>
        </xsl:otherwise>
      </xsl:choose>
    </xsl:otherwise>
  </xsl:choose>
</xsl:template>

</xsl:stylesheet>
```