User-Friendly Formatting of the Reactor ParameterList XML Files for CASL Applications

Srdjan Simunovic

[simunovics@ornl.gov](mailto:simunovics@ornl.gov)

## Summary

The ParameterList (PL) XML files are used for problem definition and initialization of DENOVO simulations. The PL XML files cannot be easily examined in their raw form, which hinders the modeling and simulation process. In order to facilitate analysis of the PL XML files, an EXtensible Stylesheet Language Transformation (XSLT) was written to transform the PL XML files into a user-friendly format and to display the transformed content in the Internet browsers. This method eliminates a need for development of a separate viewing application and takes advantage of the existing, widely used and familiar computer tools.

## 1. Introduction

The EXtensible Markup Language (XML) [1] format in general is not very convenient for direct analysis of the data it encodes. An XML file is readable but not user-friendly in its rudimentary format. The XML tags organize the encoded data and replace the visual clues that are usually employed in the display-oriented documents. At the same time, these tags introduce a considerable visual clutter for human interpretation. Internet browsers and XML editors can display the XML files in a more convenient form of trees, albeit still with all the node tags. The XML files using ParameterList (PL) specification from Trilinos Teuchos [2] library are used for specifying problem configurations for DENOVO simulations. The PL XML files are constructed from just two types of data structures (XML nodes); ParameterList and Parameter. ParameterList can contain both Parameters and ParameterLists, and is generally used for grouping data and creating data structures. Parameter XML node contains the actual data. The data in Parameter nodes is encoded into the XML node arguments. There is no text or additional XML nodes in the Parameter XML nodes. The arguments in the Parameter node are name, type and value. The data in the Parameter argument value are stored as strings representing the scalar or array data. The array data is also encoded as a string in the argument value. The array string is composed of curly braces enclosing array entries separated by commas. Therefore, access to the atomic data fields encoded in the arguments has to be accompanied by interpretation/decoding of the argument content strings.

There are numerous solutions for presenting the PL XML data in a more user-friendly form. One approach, described here, is to transform the PL XML with the EXtensible Stylesheet Language (XSL) [3] and the XSL Transformations (XSLT) [4,5]. XSL is a style sheet language for XML documents. XSLT is a declarative, template-driven language for transformation of XML documents.

In a typical business Web application, a query initiated by a user/client extracts data from a database on the server, creates an XML document, and then sends it to the user’s browser through an XSL transform (XSLT) resulting in an (X)HTML document. The XSL transform can be designed to occur on the server or on the client (browser). Mobile and cloud implementations blur this simplistic sketch, but it suffices for the current description.

If an XSL transformation can occur on the client (browser) and thereby eliminate the need for a server-based transformation, we can take advantage of the client’s functionality and view the PL XML files as we would any other local HTML document. The XSL transformation of an XML file is invoked by placing the XSL processing instruction in the header of the XML file as:

<?xml-stylesheet version="1.0" type="text/xsl" href="PL9.xsl"?>

where in this case, PL9.xsl, is an XSLT file that is used to transform the XML document by the client/browser. To avoid discussion and consideration of the security issues associated with the remote XSLT files, their execution, etc., we will assume that the XSLT file resides on the same computer as the PL XML file, and that the browser/client has typical security settings.

The client-side XSL transformation implies existence of an XSLT engine in the client/browser. The good news is that all the major browsers support XSLT. The bad news is that they only support the version 1 [4] of XSLT. The XSLT version 2 [5] has substantially larger functionality and capability compared to version 1 but it is only implemented by a few applications and libraries with various licensing restrictions. From the usability point of view, the elegance and simplicity of just opening of a PL XML file in an Internet browser trumps the modern language implementation, so that the transformation of the PL XML file has been written in the XSLT version 1 syntax. XSLT version 2 standard is backward-compatible with version 1, so that this decision does not restrict future compatibility.

## 2. Usage

In order to view the PL XML files that are transformed by the XSLT style sheet, you have to make sure that the header (two first lines) of the XML file contains the following processing instructions:

<?xml version="1.0" encoding="UTF-8"?>

<?xml-stylesheet version="1.0" type="text/xsl" href="PL9.xsl"?>

These two lines will be inserted into the PL XML file if switches --xml and --xslt were used in the reactor file parser. The path to the XSLT file, PL9.xsl, is given in the href argument and should reside on the computer where the PL XML file will be opened. If a PL XML file resides on a server, the XSLT file (PL9.xsl) should also reside on the server. As stated before, other use scenarios involve manipulation of the client and server security settings, and are not discussed in this document.

After you have verified the header of the PL XML file (e.g. head -2 file.xml) and the location of the PL9.xsl file, you can just open the file in the XSLT-enabled browser, which should display the PL XML file in a user-friendly form. A display of a typical PL XML file is shown in Figure 1:

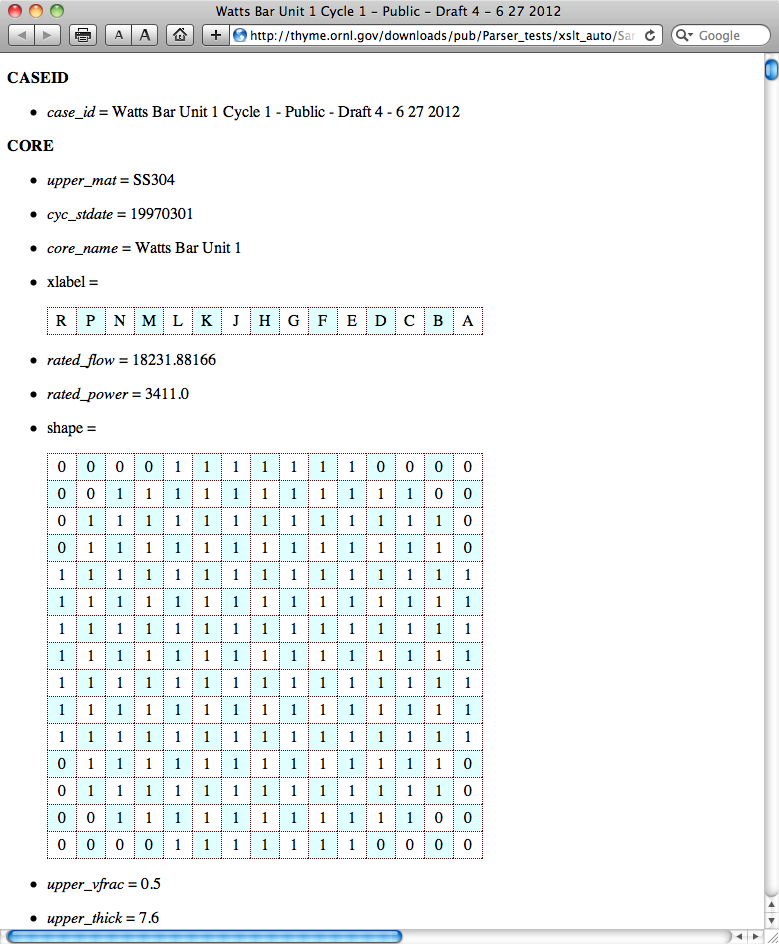


Figure 1: HTML rendering of a PL XML file using XSLT file PL9.xsl

If you get a blank page in the browser, refresh the page to force the transformation. The corresponding PL XML source can be viewed by using ‘View Source’ or similar option in a browser, and the result shown in Figure 2.

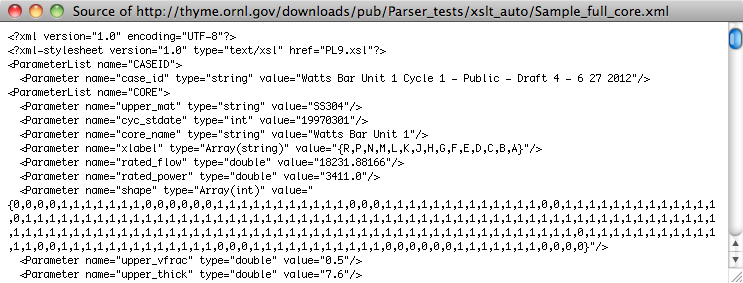


Figure 2: PL XML file transformed by the browser into HTML using the XSLT file PL9.xsl

As can be seen, the source file is the PL XML file. The file is just rendered in the browser as HTML. The current HTML display format of the PL XML files is rather simple. Much more sophisticated presentation with hyperlinks can be constructed. The most involved XSLT functions, such as decoding of maps and shapes from long strings, and transforming them into the HTML table forms, have been implemented, so that other extensions should not be too complicated.

Operating systems and browsers that were tested for the XSLT engine compatibility with the XSLT style sheet PL9.xsl are:

* Mac OS X (Firefox, Safari, Opera, Chrome),
* Windows 7 (Firefox, Internet Explorer, Chrome),
* Linux (Firefox),
* IOS (Safari, Opera).

The above list is not complete and does not indicate that, for example, the viewing of PL XML files using Chrome browser does not work on Linux. Most likely it does work, as the browser code base is usually the same across the operating systems. Opera browser is known to have problems with printing of dynamic content files.

On the CASL’s fissile four machines, you can also view the files in the terminal mode using lynx web browser. lynx does not have an XSLT engine so that the PL XML file first has to be converted into an HTML file. That can be accomplished by using command xsltproc as below (assume that PL XML file is file.xml):

xsltproc file.xml –o file.html

You can then view the resulting HTML file file.html using lynx browser as:

lynx file.html

Use control-p and control-n in lynx to scroll line by line. In xterm window of the fissile machines keyboard arrows result in page up and page down scrolling. Key 'q' will quit lynx. You can also create a text file, as:

lynx –dump file.html > file.txt

Another possibility is to use pipes:

xsltproc file.xml | lynx –stdin

and view the PL XML file without generating the intermediate html file. Note that file file.html will not be a well-formed xml file. This is because of the meta tag in HTML standard is not closed and therefore the xml parsers will report an error due to the unclosed tag.

## 3. XSLT Code Documentation

A common way to describe the XSL transformation process is as a mapping of an XML source-tree into an XML result-tree. The XML result-tree in our case is an HTML tree. The XSLT file has to be a well-formed XML document. The XML form makes the XSLT syntax appear very verbose and clumsy, but this drawback is well compensated by the benefits from the automatic inheritance of all the lexical apparatus of XML. That enables an XSLT style sheet to be an input or output of a transformation, and makes it easy to dynamically generate an XML output within the XSLT style sheet.

The XSLT is a declarative-type language that applies transformations on the data selected from the XML source-tree by pattern matching without describing its control flow. An XSLT style sheet consists of a set of template and selection rules. The order of the rule application is immaterial, and when several rules match the same input, the default or user-supplied algorithms for conflict-resolution can be applied. The idea is to describe what the transformation should do, rather than to describe how to do it. The XML and XSLT files are not processed sequentially, line by line. Each XSLT rule is applied to the nodes in the input XML tree, although the template rule can prescribe the order of the nodes to be processed. Therefore, the explanation of XSLT code below in PL9.xsl requires some familiarity with the concepts of declarative languages, rule-based processing and standard references on XSLT [6,7].

In the following, segments of the XSLT file PL9.xsl will be listed and accompanied with description of the function of each segment. Mark [clip] will indicate removed section of the XSLT code that is not relevant for the description.

The XSLT file starts with the header section that declares the document type and the namespaces that are used in the code. The standard namespaces for XSL (xsl) and XML Schema (xs) [8] are used.

<?xml version="1.0" encoding="UTF-8"?>

<xsl:stylesheet version="1.0"

xmlns:xsl="http://www.w3.org/1999/XSL/Transform"

xmlns:xs="http://www.w3.org/2001/XMLSchema"

xmlns:exslt="http://exslt.org/common"

xmlns:msxsl="urn:schemas-microsoft-com:xslt"

exclude-result-prefixes="exslt msxsl"

>

<msxsl:script language="JScript" implements-prefix="exslt">

this['node-set'] = function (x) {

return x;

}

</msxsl:script>

The namespaces msxsl (for Internet Explorer) and exslt (for all other browsers) are extensions implementing node-set() function that will be referenced below. The <msxsl:script> element is used to avoid switching in the code between different browsers. Some prefer using of <xsl:choose> tag for testing the client/browser configuration and picking the corresponding namespace and the function, but given that we are using only node-set() function and that it has the identical name in both implementations, this approach should suffice.

The template in the XSLT file that matches the root of the PL XML document, always evaluates as true and initiates the entire transformation:

<xsl:template match="/">

<xsl:element name="html">

<xsl:element name="head">

<xsl:element name="title">

<xsl:value-of select="ParameterList/Parameter[@name='case\_id']/@value" />

</xsl:element>

<!-- CSS 3, IE will not work -->

<style type="text/css">

**[clip]**

</style>

</xsl:element>

<xsl:element name="body">

**<xsl:element name="p">**

**<strong><xsl:value-of select="ParameterList/@name" /></strong>**

**</xsl:element>**

**<xsl:element name="ul">**

**<xsl:apply-templates select="ParameterList/Parameter" />**

**</xsl:element>**

**<xsl:apply-templates select="ParameterList/ParameterList" />**

</xsl:element>

</xsl:element>

</xsl:template>

The title for the html document is set from the case\_id entry in the PL XML file. Style section uses Cascading Style Sheets (CSS) [9] level 3 for presentation semantics of the html content. The content of this segment is used to create checkerboard coloring for the tables in order to ease the viewing of patterns in the core and assembly maps. Other CSS styling commands can be added to the same section as needed. With the root of the PL XML document matched, we can now initiate templates of all other entries. The fact that the PL XML file only has two types of entries, ParameterList and Parameter, enormously simplifies the processing and allows us to use recursion. Each ParameterList makes a new paragraph element that holds ParameterList node name in boldface font style. It is followed by unnumbered list <ul> entries of Parameter, and ParameterList nodes. Indentation in the HTML is due to paragraph and list entries, and it depicts hierarchy in the PL XML data. Note that there is an exception for the root element in the PL XML file, CASEID, which is displayed in the HTML as a sibling to other main blocks (CORE, ASSEMBLIES, etc.). This was done to make the HTML file similar to the reactor input file where the [CASEID] block is on the same level as the other main blocks. To show the actual hierarchy of the PL XML file and the root element, you can simply delete the boldface region in the above listing and replace it with:

<xsl:apply-templates select="\*"/>

so that the body element looks like:

<xsl:element name="body">

<xsl:apply-templates select="\*"/>

</xsl:element>

With the main match initiated, we now define transformation rules for the nodes that match the two main types of nodes, ParameterList and Parameter. The rule for ParameterList is simple; it just creates an unnamed list for Parameter nodes and initiates recursion for ParameterList nodes.

<xsl:template match="ParameterList">

<xsl:element name="p">

<xsl:element name="strong">

<xsl:value-of select="@name" />

</xsl:element>

</xsl:element>

<xsl:element name="ul">

<xsl:apply-templates select="Parameter" />

</xsl:element>

<xsl:element name="ul">

<xsl:apply-templates select="ParameterList" />

</xsl:element>

</xsl:template>

The template and its rules for Parameter nodes are more complex. They have to distinguish between scalars, arrays, assembly maps, and core maps. Since we have to work with the XSLT version 1, many elementary functions have to be explicitly implemented. The Parameter match rule is selected using <xsl:choose> node:

<xsl:template match="Parameter">

<xsl:choose>

<xsl:when test="@name = 'shape' or @name = 'dancoff\_map' or @name = 'cell\_map'">

[clip]

</xsl:when>

<xsl:when test="@name = 'insert\_map' or @name = 'assm\_map' or @name = 'crd\_map' or @name = 'det\_map' or @name = 'crd\_bank'">

[clip]  
 </xsl:when>

<xsl:when test="not(@name = 'shape' or @name = 'dancoff\_map' or @name = 'cell\_map' or @name = 'insert\_map' or @name = 'assm\_map' or @name = 'crd\_map' or @name = 'det\_map' or @name = 'crd\_bank') and starts-with(@type,'Array')">

[clip]

</xsl:when>

<xsl:otherwise>

[clip]

</xsl:otherwise>

</xsl:choose>

</xsl:template>

where code segments that to be applied when the conditions are met were omitted for clarity. The first condition (the first <xsl:when> block) selects the full square maps that are to be displayed as square tables. The second condition selects maps for square table display that need to be combined with the core shape map because their entries are specified only for the entries in the core shape that are 1. The third condition selects the regular arrays that are to be displayed as one-dimensional tables. If none of these conditions are met (<xsl:otherwise>), the Parameter node is scalar.

For the full, two-dimensional maps, two variables are defined, num\_pins and map, that hold the dimension of the map and the map data string, respectively. If num\_pins variable is greater than 0, the a new template map\_map is invoked, with variables map and num\_pins as:

<xsl:if test="$num\_pins > 0">

<xsl:element name="li">

<xsl:element name="p">

<xsl:value-of select="@name" /> =

<xsl:call-template name="make\_map">

<xsl:with-param name="num\_pins">

<xsl:value-of select="$num\_pins" />

</xsl:with-param>

<xsl:with-param name="map">

<xsl:value-of select="$map" />

</xsl:with-param>

</xsl:call-template>

</xsl:element>

</xsl:element>

</xsl:if>

A similar rule applies for processing of the reactor maps that depend on the core shape data, except that now a different mapping template is invoked and the core shape (shape) variable is passed together with the core size (num\_pins) and the data (map) string:

<xsl:if test="$num\_pins > 0">

<xsl:element name="li">

<xsl:element name="p">

<xsl:value-of select="@name" /> =

<xsl:call-template name="make\_core\_map">

<xsl:with-param name="num\_pins">

<xsl:value-of select="$num\_pins" />

</xsl:with-param>

<xsl:with-param name="shape">

<xsl:value-of select="$shape" />

</xsl:with-param>

<xsl:with-param name="map">

<xsl:value-of select="$map" />

</xsl:with-param>

</xsl:call-template>

</xsl:element>

</xsl:element>

</xsl:if>

Processing of the regular arrays starts with removing of the curly braces from the data string (translate() function) and the resulting string is split (tokenize template [7]) into a tree fragment of nodes with the tag <token>. This process is used across the templates for Parameter nodes that contain Array strings. The exslt:node-set() function converts the result tree fragment into a node-set which is the reason for the inclusion of its name space in the header. The corresponding XSLT match entry for the regular arrays is shown below.

<xsl:when test="not(@name = 'shape' or @name = 'dancoff\_map' or @name = 'cell\_map' or @name = 'insert\_map' or @name = 'assm\_map' or @name = 'crd\_map' or @name = 'det\_map' or @name = 'crd\_bank') and starts-with(@type,'Array')">

<xsl:variable name="var\_ma" select="translate(@value,'\{\}','')" />

<xsl:variable name="map\_array">

<xsl:call-template name="tokenize">

<xsl:with-param name="string" select="$var\_ma"/>

<xsl:with-param name="delimiters" select="','"/>

</xsl:call-template>

</xsl:variable>

<xsl:element name="li">

<xsl:element name="p">

<xsl:value-of select="@name" /> =

<table>

**<tr>**

<xsl:for-each select="exslt:node-set($map\_array)/token">

<td>

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

</td>

</xsl:for-each>

**</tr>**

</table>

</xsl:element>

</xsl:element>

</xsl:when>

A horizontal, one-dimensional table is generated from the tokenized array elements. If a vertical presentation of the HTML table is preferred, the boldface <tr> element tags just need to be moved to be immediately enclosing the <td> element above.

Display of scalar Parameter node is straightforward operation that is enclosed in <xsl:otherwise> element.

The XSLT template rules of some complexity are make\_map and make\_core\_map. They both invoke another template, disp\_nrow. This template is written as an elementary operation that displays a list of elements into a square HTML table. The make\_map template just tokenizes the input data string and passes it to the disp\_nrow template. The make\_core\_map first creates a node list of the size of the core shape and inserts the input data into locations where the core shape list value is equal to 1. This new list is then passed into disp\_nrow template. Additional templates for summation of a node list (sum) and creation of a number sequence (number.options) are used in the make\_map and make\_core\_map templates.

The XSLT file is about 450 lines long. It is relatively small mostly due to strategy of using call-template style of XSLT programming. Looking back, it may be updated to make it less error prone and more humanly readable. Several segments were cut-and-pasted repeatedly and may be consolidated into separate templates. While this would not necessarily reduce the size, it would make the code more robust and portable. However, that can be postponed until new functionality of display style is needed.

## 4. Conclusion

A simple XSLT style sheet for the PL XML files have been written in order to display the PL XML data in a user-friendly form. The sheet is very simple and can be extended to include additional elements such as hyperlinks and additional transformations. A perhaps useful extension would be to develop an XSLT style sheet and an XML Schema [8] for the PL XML validation, which would check the validity of the data in the PL XML files from different application perspectives, such as neutronics or thermal fluid flow simulation, and display possible errors. The XSLT route may also be a feasible solution for transformation between different PL XML files and their corresponding applications.

## References

1. World Wide Web Consortium (W3C), Extensible Markup Language (XML), <http://www.w3.org/XML/>.
2. Teuchos, <http://trilinos.sandia.gov/packages/teuchos/>.
3. W3C, The Extensible Stylesheet Language Family (XSL), http://www.w3.org/Style/XSL/.
4. W3C, XSL Transformations (XSLT) Version 1.0, <http://www.w3.org/TR/xslt/>, W3C Recommendation, 1999.
5. W3C, XSL Transformations (XSLT) Version 2.0, <http://www.w3.org/TR/xslt20/>, W3C Recommendation, 2007.
6. Tidwell, D., XSLT: Mastering XML Transformations, O’Reilly, 2001.
7. Mangano, S., XSLT Cookbook, 2nd Ed., O’Reilly, 2006.
8. Walmsley, P., Definitive XML Schema, Prentice Hall, 2001.
9. W3C, Cascading Style Sheets (CSS) Snapshot 2010, <http://www.w3.org/TR/CSS/>, W3C Working Group Note 12 May 201, 2011.

## Appendix

Listing of the PL9.xsl file

<?xml version="1.0" encoding="UTF-8"?>

<xsl:stylesheet version="1.0"

xmlns:xsl="http://www.w3.org/1999/XSL/Transform"

xmlns:xs="http://www.w3.org/2001/XMLSchema"

xmlns:exslt="http://exslt.org/common"

xmlns:msxsl="urn:schemas-microsoft-com:xslt"

exclude-result-prefixes="exslt msxsl"

>

<!--

version 9.C:07/27/2012:463:13911:Srdjan Simunovic

simunovics@ornl.gov

http://thyme.ornl.gov/simunovics

XSLT for ParameterList reactor file

-->

<msxsl:script language="JScript" implements-prefix="exslt">

this['node-set'] = function (x) {

return x;

}

</msxsl:script>

<xsl:output method="html"

doctype-system="http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd"

doctype-public="-//W3C//DTD XHTML 1.0 Transitional//EN" />

<!-- SSMOD: Note to self, change tags to xsl elements -->

<xsl:template match="/">

<xsl:element name="html">

<xsl:element name="head">

<xsl:element name="title">

<xsl:value-of select="ParameterList/Parameter[@name='case\_id']/@value" />

</xsl:element>

<!-- CSS 3, IE will not work -->

<style type="text/css">

table, td

{

border-color: #600;

border-style: dotted;

}

table

{

border-width: 0 0 1px 1px;

border-spacing: 0;

border-collapse: collapse;

}

td

{

margin: 0;

padding: 4px;

border-width: 1px 1px 0 0;

width: 20px;

text-align: center;

}

tr:nth-child(even) td:nth-child(odd)

{

background-color: #E0FFFF;

}

tr:nth-child(odd) td:nth-child(even)

{

background-color: #E0FFFF;

}

</style>

</xsl:element>

<xsl:element name="body">

<!-- comment from here to -A1- if you want the exact hierarchy -->

<xsl:element name="p">

<strong><xsl:value-of select="ParameterList/@name" /></strong>

</xsl:element>

<xsl:element name="ul">

<xsl:apply-templates select="ParameterList/Parameter" />

</xsl:element>

<xsl:apply-templates select="ParameterList/ParameterList" />

<!-- -A1- -->

<!-- uncomment if you want the exact hierarchy

<xsl:apply-templates select="\*"/>

-->

</xsl:element>

</xsl:element>

</xsl:template>

<xsl:template match="ParameterList">

<xsl:element name="p">

<xsl:element name="strong">

<xsl:value-of select="@name" />

</xsl:element>

</xsl:element>

<xsl:element name="ul">

<xsl:apply-templates select="Parameter" />

</xsl:element>

<xsl:element name="ul">

<xsl:apply-templates select="ParameterList" />

</xsl:element>

</xsl:template>

<xsl:template match="Parameter">

<!-- SSMOD: Modify to explicit checking of each keyword -->

<xsl:choose>

<xsl:when test="@name = 'shape' or @name = 'dancoff\_map' or @name = 'cell\_map'">

<xsl:variable name="num\_pins">

<xsl:choose>

<xsl:when test="../../../Parameter[@name = 'num\_pins']">

<xsl:value-of select="../../../Parameter[@name = 'num\_pins']/@value" />

</xsl:when>

<xsl:when test="../Parameter[@name = 'num\_pins']">

<xsl:value-of select="../Parameter[@name = 'num\_pins']/@value" />

</xsl:when>

<xsl:when test="../Parameter[@name = 'core\_size']">

<xsl:value-of select="../Parameter[@name = 'core\_size']/@value" />

</xsl:when>

<xsl:otherwise>

<xsl:value-of select="0" />

</xsl:otherwise>

</xsl:choose>

</xsl:variable>

<xsl:variable name="map" select="@value" />

<xsl:if test="$num\_pins > 0">

<xsl:element name="li">

<xsl:element name="p">

<xsl:value-of select="@name" /> =

<xsl:call-template name="make\_map">

<xsl:with-param name="num\_pins">

<xsl:value-of select="$num\_pins" />

</xsl:with-param>

<xsl:with-param name="map">

<xsl:value-of select="$map" />

</xsl:with-param>

</xsl:call-template>

</xsl:element>

</xsl:element>

</xsl:if>

</xsl:when>

<!-- maps over templates -->

<xsl:when test="@name = 'insert\_map' or @name = 'assm\_map' or @name = 'crd\_map' or @name = 'det\_map' or @name = 'crd\_bank'">

<xsl:variable name="num\_pins" select="../Parameter[@name = 'core\_size']/@value" />

<xsl:variable name="shape" select="../Parameter[@name = 'shape']/@value" />

<xsl:variable name="map" select="@value" />

<xsl:if test="$num\_pins > 0">

<xsl:element name="li">

<xsl:element name="p">

<xsl:value-of select="@name" /> =

<xsl:call-template name="make\_core\_map">

<xsl:with-param name="num\_pins">

<xsl:value-of select="$num\_pins" />

</xsl:with-param>

<xsl:with-param name="shape">

<xsl:value-of select="$shape" />

</xsl:with-param>

<xsl:with-param name="map">

<xsl:value-of select="$map" />

</xsl:with-param>

</xsl:call-template>

</xsl:element>

</xsl:element>

</xsl:if>

</xsl:when>

<!-- regular arrays -->

<xsl:when test="not(@name = 'shape' or @name = 'dancoff\_map' or @name = 'cell\_map' or @name = 'insert\_map' or @name = 'assm\_map' or @name = 'crd\_map' or @name = 'det\_map' or @name = 'crd\_bank') and starts-with(@type,'Array')">

<xsl:variable name="var\_ma" select="translate(@value,'\{\}','')" />

<xsl:variable name="map\_array">

<xsl:call-template name="tokenize">

<xsl:with-param name="string" select="$var\_ma"/>

<xsl:with-param name="delimiters" select="','"/>

</xsl:call-template>

</xsl:variable>

<xsl:element name="li">

<xsl:element name="p">

<xsl:value-of select="@name" /> =

<table>

<tr>

<xsl:for-each select="exslt:node-set($map\_array)/token">

<td>

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

</td>

</xsl:for-each>

</tr>

</table>

</xsl:element>

</xsl:element>

</xsl:when>

<!-- scalars -->

<xsl:otherwise>

<xsl:element name="li">

<xsl:element name="p">

<xsl:element name="i">

<xsl:value-of select="@name" />

</xsl:element> = <xsl:value-of select="@value" />

</xsl:element>

</xsl:element>

</xsl:otherwise>

</xsl:choose>

</xsl:template>

<!-- vanilla map -->

<xsl:template name="make\_map">

<xsl:param name="num\_pins"></xsl:param>

<xsl:param name="map"></xsl:param>

<xsl:variable name="var\_ma" select="translate($map,'\{\}','')" />

<xsl:variable name="map\_array">

<xsl:call-template name="tokenize">

<xsl:with-param name="string" select="$var\_ma"/>

<xsl:with-param name="delimiters" select="','"/>

</xsl:call-template>

</xsl:variable>

<xsl:call-template name="disp\_nrow">

<xsl:with-param name="dat" select="exslt:node-set($map\_array)/token" />

<xsl:with-param name="num\_pins" select="$num\_pins" />

</xsl:call-template>

</xsl:template>

<!-- map of maps -->

<xsl:template name="make\_core\_map">

<xsl:param name="num\_pins"></xsl:param>

<xsl:param name="shape"></xsl:param>

<xsl:param name="map"></xsl:param>

<xsl:variable name="var\_sh" select="translate($shape,'\{\}','')" />

<xsl:variable name="sh\_array">

<xsl:call-template name="tokenize">

<xsl:with-param name="string" select="$var\_sh"/>

<xsl:with-param name="delimiters" select="','"/>

</xsl:call-template>

</xsl:variable>

<xsl:variable name="shape\_array" select="exslt:node-set($sh\_array)/token" />

<xsl:variable name="var\_ma" select="translate($map,'\{\}','')" />

<xsl:variable name="map\_array">

<xsl:call-template name="tokenize">

<xsl:with-param name="string" select="$var\_ma"/>

<xsl:with-param name="delimiters" select="','"/>

</xsl:call-template>

</xsl:variable>

<xsl:variable name="map\_map">

<xsl:for-each select="$shape\_array">

<xsl:variable name="ips" select="position()"/>

<xsl:variable name="sli" select="$shape\_array[position() &gt;= 1 and position() &lt;= $ips]"/>

<xsl:variable name="smi" select="sum($sli)" />

<xsl:variable name="smv" select="exslt:node-set($map\_array)/token[$smi]"/>

<xsl:element name="ent">

<xsl:choose>

<xsl:when test="number(.) = 0">

<xsl:value-of select="''" />

</xsl:when>

<xsl:otherwise>

<xsl:value-of select="$smv" />

</xsl:otherwise>

</xsl:choose>

</xsl:element>

</xsl:for-each>

</xsl:variable>

<xsl:call-template name="disp\_nrow">

<xsl:with-param name="dat" select="exslt:node-set($map\_map)/ent" />

<xsl:with-param name="num\_pins" select="$num\_pins" />

</xsl:call-template>

</xsl:template>

<!-- cookbook stuff -->

<xsl:template name="tokenize">

<xsl:param name="string" select="''" />

<xsl:param name="delimiters" select="' &#x9;&#xA;'" />

<xsl:choose>

<xsl:when test="not($string)" />

<xsl:when test="not($delimiters)">

<xsl:call-template name="\_tokenize-characters">

<xsl:with-param name="string" select="$string" />

</xsl:call-template>

</xsl:when>

<xsl:otherwise>

<xsl:call-template name="\_tokenize-delimiters">

<xsl:with-param name="string" select="$string" />

<xsl:with-param name="delimiters" select="$delimiters" />

</xsl:call-template>

</xsl:otherwise>

</xsl:choose>

</xsl:template>

<!--

<xsl:template name="\_tokenize-characters">

<xsl:param name="string" />

<xsl:if test="$string">

<token><xsl:value-of select="substring($string, 1, 1)" /></token>

<xsl:call-template name="\_tokenize-characters">

<xsl:with-param name="string" select="substring($string, 2)" />

</xsl:call-template>

</xsl:if>

</xsl:template>

-->

<xsl:template name="\_tokenize-characters">

<xsl:param name="string" />

<xsl:param name="len" select="string-length($string)"/>

<xsl:choose>

<xsl:when test="$len = 1">

<token><xsl:value-of select="$string"/></token>

</xsl:when>

<xsl:otherwise>

<xsl:call-template name="\_tokenize-characters">

<xsl:with-param name="string" select="substring($string, 1, floor($len div 2))" />

<xsl:with-param name="len" select="floor($len div 2)"/>

</xsl:call-template>

<xsl:call-template name="\_tokenize-characters">

<xsl:with-param name="string" select="substring($string, floor($len div 2) + 1)" />

<xsl:with-param name="len" select="ceiling($len div 2)"/>

</xsl:call-template>

</xsl:otherwise>

</xsl:choose>

</xsl:template>

<xsl:template name="\_tokenize-delimiters">

<xsl:param name="string" />

<xsl:param name="delimiters" />

<xsl:param name="last-delimit"/>

<xsl:variable name="delimiter" select="substring($delimiters, 1, 1)" />

<xsl:choose>

<xsl:when test="not($delimiter)">

<token><xsl:value-of select="$string"/></token>

</xsl:when>

<xsl:when test="contains($string, $delimiter)">

<xsl:if test="not(starts-with($string, $delimiter))">

<xsl:call-template name="\_tokenize-delimiters">

<xsl:with-param name="string" select="substring-before($string, $delimiter)" />

<xsl:with-param name="delimiters" select="substring($delimiters, 2)" />

</xsl:call-template>

</xsl:if>

<xsl:call-template name="\_tokenize-delimiters">

<xsl:with-param name="string" select="substring-after($string, $delimiter)" />

<xsl:with-param name="delimiters" select="$delimiters" />

</xsl:call-template>

</xsl:when>

<xsl:otherwise>

<xsl:call-template name="\_tokenize-delimiters">

<xsl:with-param name="string" select="$string" />

<xsl:with-param name="delimiters" select="substring($delimiters, 2)" />

</xsl:call-template>

</xsl:otherwise>

</xsl:choose>

</xsl:template>

<!--

sum recursion

SSMOD: remember error in using select in variable name tag

-->

<xsl:template name="sum">

<xsl:param name="nodes" select="/.."/>

<xsl:param name="result" select="'0'"/>

<xsl:choose>

<xsl:when test="not($nodes)">

<xsl:value-of select="$result"/>

</xsl:when>

<xsl:otherwise>

<!--

call or apply template that will determine value of node

unless the node is literally the value to be summed

-->

<xsl:variable name="value">

<xsl:value-of select="$nodes[1]" />

<!--

<xsl:call-template name="some-function-of-a-node">

<xsl:with-param name="node" select="$nodes[1]"/>

</xsl:call-template>

-->

</xsl:variable>

<xsl:call-template name="sum">

<xsl:with-param name="nodes" select="$nodes[position() != 1]"/>

<xsl:with-param name="result" select="$result + $value"/>

</xsl:call-template>

</xsl:otherwise>

</xsl:choose>

</xsl:template>

<xsl:template name="disp\_nrow">

<xsl:param name="dat"></xsl:param>

<xsl:param name="num\_pins"></xsl:param>

<xsl:variable name="idx">

<xsl:for-each select="$dat">

<xsl:element name="val">

<xsl:attribute name="row">

<xsl:value-of select="ceiling(position() div $num\_pins)" />

</xsl:attribute>

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

</xsl:element>

</xsl:for-each>

</xsl:variable>

<!-- create number sequence -->

<xsl:variable name="row\_list">

<xsl:call-template name="number.options">

<xsl:with-param name="i" select="number(1)" />

<xsl:with-param name="count" select="number($num\_pins)" />

</xsl:call-template>

</xsl:variable>

<xsl:element name="table">

<xsl:for-each select="exslt:node-set($row\_list)/num">

<xsl:element name="tr">

<xsl:for-each select="exslt:node-set($idx)/val[@row = current()]">

<xsl:element name="td">

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

</xsl:element>

</xsl:for-each>

</xsl:element>

</xsl:for-each>

</xsl:element>

</xsl:template>

<!-- create number sequence i to count -->

<xsl:template name="number.options">

<xsl:param name="i" />

<xsl:param name="count" />

<xsl:if test="$i &lt;= $count">

<xsl:element name="num">

<xsl:value-of select="$i"/>

</xsl:element>

</xsl:if>

<xsl:if test="$i &lt;= $count">

<xsl:call-template name="number.options">

<xsl:with-param name="i" select="$i + 1"/>

<xsl:with-param name="count" select="$count"/>

</xsl:call-template>

</xsl:if>

</xsl:template>

</xsl:stylesheet>