# Automated mass production of XSLT stylesheets

Bob **DuCharme**

## Abstract

Many have wished for a tool that would automate the creation of XSLT stylesheets. Building the interface alone to such a tool sounds like a tough job, and getting it to output working XSLT stylesheets that accomplish non-trivial tasks also sounds challenging. However, the comfort level of nearly all computer users with basic spreadsheet software actually makes the first task simpler than it once appeared to be, and the ease with which popular spreadsheet programs now save their contents in XML means that when you start with the right spreadsheet template, an XSLT stylesheet is not difficult to create from the XML version of a spreadsheet that uses that template.

# Table of Contents

# 1. Introduction

Ever since XSLT became a standard in 2001, people have wondered about the possibility of tools that would automate the creation of XSLT stylesheets. They pictured a tool that would let users specify transformation tasks to perform without requiring those users to understand XSLT syntax. While tools such as XSLMaker [http://www.xslmaker.com/mainPage-2.html.xml], Stylus Studio's XSLT Designer [http://www.stylusstudio.com/xslt_designer.html] and Axizon's Tiger XSLT mapper [http://www.axizon.com/] promise "drag and drop" creation of XSLT stylesheets, the most successful tools to help create XSLT stylesheets are ultimately IDEs that colorize syntax and automate syntax checking and typing tasks. Complex XSLT transformations would be difficult to express in a graphical user interface, and simple tasks, such as the deletion, renaming, and reordering of elements, are really not that difficult to code in XSLT.

Just the design of the graphical user interface that let someone express transformations would be a huge task. Will the users be transforming many input formats into a small number of common output formats, such as HTML and XSL-FO? Will they be converting a limited number of input formats, as expressed by a set of in-house schemas, to a wide variety of output formats for different media? Do they want to rearrange one business partner's XML to conform with the schema of another business partner for transaction-oriented XML, with no thought of publishing that XML in a readable medium? No single user interface will be ideal for all these classes of XSLT transformations, and designing such an interface, testing it, and training people to use it is only half the work necessary (well, maybe two-thirds)--the tool must still output working XSLT that expresses the user's wishes. The W3C XForms standard holds promise for easing the development of an interface that can then output XSLT stylesheets, but my own research so far shows that while the XForms Recommendation [XFORMS] provides enough syntax to express a GUI interface that can create working XSLT stylesheets, no XForms implementation is yet mature enough to implement this. [1]

There's one interface that nearly all computer users understand, although its suitability to this problem didn't occur to me until recently: electronic spreadsheets. The Visicalc spreadsheet program was the legendary "killer app" that sold the majority of early 8-bit microcomputers into office environments, the Lotus 1-2-3 spreadsheet triggered the purchase of most of the first generation of 16-bit PCs, especially those from IBM, and Excel continues to be one of Microsoft's most successful products.

Besides the departmental budgets and business plan applications that let you enter certain numbers into spreadsheets and see others appear automatically because of the mathematical expressions stored in those spreadsheets, many people use spreadsheets as a freeform database. For example, if a business analyst's boss requests a report on how the company's eight offices handle a certain task, a typical business analyst will create a new spreadsheet, add a row for each company office, and then add a column for each piece of information being tracked about the offices. It's easy enough to add, delete, and reorder columns as the analyst changes his or mind about which information to track, and the ability to set fonts and colors can turn a simple dump of the spreadsheet to a printer into an apparently well-organized report on research progress.

Such an analyst might even enter the headers across the top row, leave the rest blank, and then send copies of the spreadsheet to an assistant or to the individual offices with instructions to fill the rows out. In other words, spreadsheets let someone with no programming background create a form to fill out and accumulate information entered by others in a way that allows automated manipulation on a computer. Don't forget the perspective of the person on the receiving end of this electronic form, either--they need no new software or training to enter this data; it's reasonable for the sender to assume that the recipient will know what to do with it.

This familiarity by virtually all computer users with the use of a spreadsheet with predefined headers as a data entry tool gets us two-thirds of the way to the XSLT stylesheet generator: with headers describing various transformation

[1] A posting [http://lists.w3.org/Archives/Public/www-forms/2003Nov/0008.html] on the www-forms mailing list and on at least one mailing list devoted to a specific XForms implementation never got a satisfactory answer. I hoped to have the clicking of one button add one type of `xsl:template` element to the output and the clicking of another add a different `xsl:template` element to the output following another pattern. (I renamed the elements in my posting's example to generalize the problem description.)

tasks across the top row of a spreadsheet, a user can enter the names of elements and attributes for which tasks should be performed in the appropriate spreadsheet columns.

The native XML format of OpenOffice's Calc spreadsheet program and the increasing XML support of Microsoft Excel cover much of the remaining third of the work necessary to create our stylesheet generator. With the entered information stored in an XML representation of a spreadsheet, an XSLT stylesheet can pull that information out and use it to create a new stylesheet that performs the transformations expressed by the user who entered the element and attribute names in the spreadsheet columns.

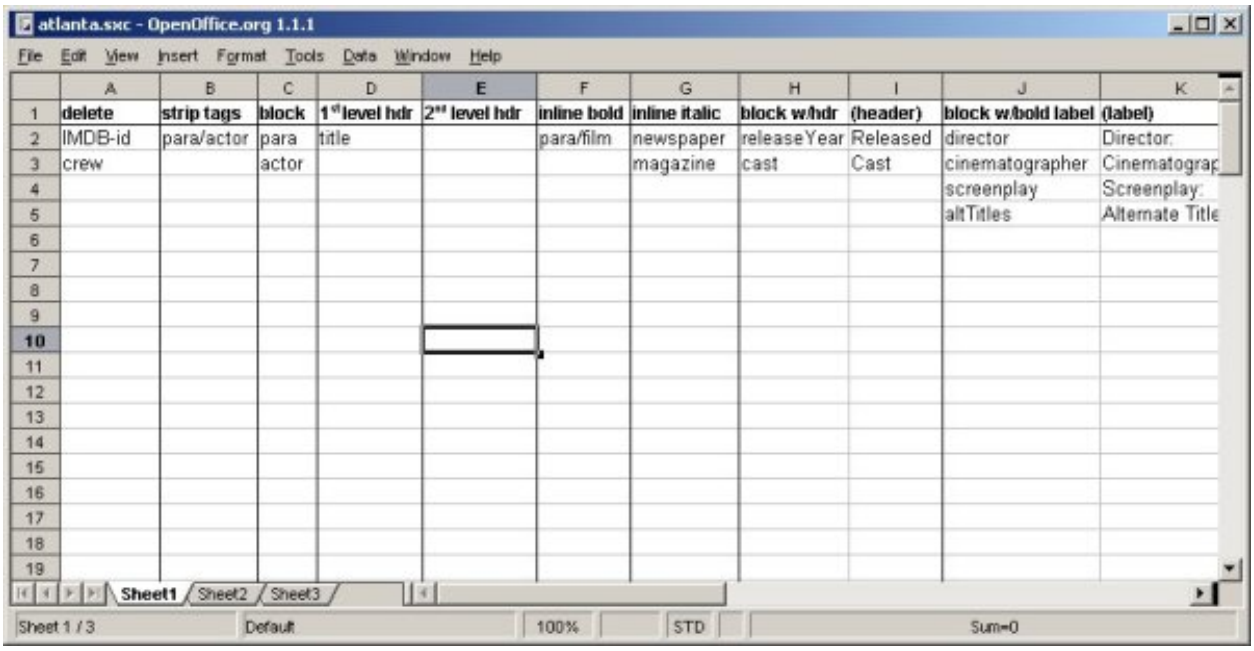
# 2. Expressing XSLT tasks on a spreadsheet

The set of tasks to list on a spreadsheet for later conversion to a stylesheet depends on the users filling out the spreadsheet, the nature and variety of the potential inputs, and the nature and variety of the potential outputs. For the use case represented by my demo, I assumed that the spreadsheet would be filled out by users who are familiar with XML but not XSLT and who need to convert different inputs to a visual layout that will end up in both HTML and PDF files. (For the latter, the system creates XSL-FO files.) The following shows a sample input document (all files referenced in this article are available at http://www.snee.com/xml/xml2005/xls2xsl.zip):

```
<movie>
  <title>Citizen Kane</title>
  <IMDB-id>tt0033467</IMDB-id>
  <picture url="http://www.imdb.com/title/tt0033467/photogallery"/>
  <altTitles>
    <altTitle>The Rosebud Movie</altTitle>
    <altTitle>CF Kane</altTitle>
    <altTitle>Mr. Newspaper</altTitle>
    <altTitle>Endorra's Revenge</altTitle>
  </altTitles>
  <releaseYear>1941</releaseYear>
  <director>Orson Wells</director>
  <cinematographer>Gregg Toland</cinematographer>
  <screenplay>Herman J. Mankiewicz and Orson Welles</screenplay>
  <cast>
    <actor>Orson Wells</actor>
    <actor>Agnes Moorhead</actor>
    <actor>Joseph Cotten</actor>
    <actor>others</actor>
  </cast>
  <crew>
    <makeup>Mel Berns</makeup>
    <costumes>Edward Stevenson</costumes>
    <specialEffects>Vernon L. Walker</specialEffects>
  </crew>
  <review>
    <para>Upon its release, the <newspaper>New York Times</newspaper>
    called <film>Citizen Kane</film> "really, really, really great."
    They didn't really, I just made that up. This is just dummy
    text. So is the next paragraph.</para>
    <para><film>Citizen Kane</film> was <actor>Agnes
    Moorehead</actor>'s first film. Wasn't she a hilarious in
    <tvshow>Bewitched</tvshow>? <magazine>Entertainment
    Tonight</magazine> called the <releaseYear>2005</releaseYear>
```

```
    <actor>Will Farrell</actor> movie <film>Bewitched</film> "OK in
    places." Actually, I made that up too.</para>
  </review>
</movie>
```

The following shows the first eleven columns of an OpenOffice spreadsheet designed for these publishing-oriented users who need to convert the XML above to web pages and Acrobat files. Entries are filled out with formatting specifications for elements from the `movie` document shown above:




| | A | B | C | D | E | F | G | H | I | J | K |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | delete | strip tags | block | 1st level hdr | 2nd level hdr | inline bold | inline italic | block w/hdr | (header) | block w/bold label | (label) |
| 2 | IMDB-id | para/actor | para | title | | para/film | newspaper | releaseYear | Released | director | Director: |
| 3 | crew | | actor | | | | magazine | cast | Cast | cinematographer | Cinematograp |
| 4 | | | | | | | | | | screenplay | Screenplay: |
| 5 | | | | | | | | | | altTitles | Alternate Title |

The first eleven columns of a spreadsheet specifying what to do to which elements of a document type. Columns A - G use one column each to specify instructions. Column H uses column I to specify a parameter, and column J uses column K for the same purpose.

**Figure 1. OpenOffice spreadsheet used for specifying a transformation**

The spreadsheet has the following columns (not all of which are shown in the illustration above) to indicate what to do with the source document type's elements:

| | |
|---|---|
| delete | Delete the elements. |
| strip tags | Pass along the elements without their enclosing tags. The only entry in this column in the illustration above is "para/actor", which demonstrates that an XPath expression can be used to identify a specific context for an element--in this case, to indicate that only `actor` elements that are children of `para` elements should have their tags stripped. |
| block | Convert to a simple block in the output, as opposed to an inline element. |
| 1st level hdr | Convert the element to a first level header. |
| 2nd level hdr | Convert the element to a second level header. |
| inline bold | Convert the element to an inline element in a bold typeface. |

| | |
|---|---|
| inline italic | Convert the element to an italicized inline element. |
| block w/hdr | Convert the element to a block element with the specified header. To represent an action that requires any parameters (in this case, the header to output above the block) a spreadsheet only needs a column for each parameter, so the spreadsheet above has a column labeled "block w/hdr" where the name of an element to format this way is specified and another column labeled "(header)" specifies the header to output above the block. This spreadsheet is formatted with vertical lines between actions, but not between the columns of a multi-column action, and parameter titles are shown in parentheses. Of course, you can format your spreadsheet in whatever way best suits your users. |
| block w/bold label | Block with a bold label: print the text as a block, with the specified text as a bolded inline label beginning the block. |
| serial list | Output this element and its siblings of the same name as an inline, comma-delimited list with the string " and " before the last sibling. |
| enclose | Output the content named in column M surrounded by the text shown on the same row in columns N and O. The filled-out spreadsheet shows that a `tvshow` element will be output surrounded by quotation marks, and that the `release` child of a `para` element should be output inside of parentheses. |
| customized | Outputs template rule stubs for each element in a separate stylesheet where hand-coded XSLT instructions can be added. |

This last column allows for the possibility of processing that can't be automated by the spreadsheet-to-stylesheet system. This enables more complex XSLT development to be incorporated into the automated system. The templates rule stubs are output to a separate stylesheet called `custElements.xsl`, and the generated stylesheet has an `xsl:include` instruction to incorporate this file's instructions. The task of filling in the stubs can be handed off to an XSLT specialist, but specialists themselves may be interested in using the whole spreadsheet-to-stylesheet tool: if they have enough stylesheets to write, a speedy way to generate the simpler code will leave them more time for the complex code.

# 3. Turning the spreadsheet into a stylesheet

My demo included the generation of working spreadsheets from both OpenOffice and Excel versions of the spreadsheet shown above. The following is an excerpt from the `content.xml` file unzipped from the sxc file that is OpenOffice 1.1.1's native format for spreadsheets, with the PCDATA bolded:

```
<table:table table:name="Sheet1" table:style-name="ta1">
  <!-- table:table-column elements omitted -->
  <table:table-row table:style-name="ro1">
    <table:table-cell table:style-name="ce1">
      <text:p>delete</text:p>
    </table:table-cell>
    <table:table-cell table:style-name="ce3">
      <text:p>strip tags</text:p>
    </table:table-cell>
    <!-- row's remaining table-cell elements omitted -->
  </table:table-row>
  <table:table-row table:style-name="ro2">
    <table:table-cell table:style-name="ce2">
      <text:p>IMDB-id</text:p>
```

```
    </table:table-cell>
    <table:table-cell>
      <text:p>para/actor</text:p>
    </table:table-cell>
```

As the screenshot above showed, this spreadsheet was filled out to indicate that the generated stylesheet should delete `IMDB-id` and `crew` elements when creating a result document. This means that the stylesheet generated from this spreadsheet should have a template rule like this:

```
<xsl:template match="IMDB-id|crew"/>
```

The generating stylesheet (that is, the one that creates the generated stylesheet that converts `movie` documents to HTML or XSL-FO files) only needs to create an `xsl:template` element with a `match` attribute and then, for that attribute's value, write out a pipe-delimited list of each value from the spreadsheet's "delete" column until it finds the blank entry at cell A4. In XPath terms, the generating stylesheet must grab each `table:table-row/table:table-cell[1]/text:p` element in the table starting at the second row (to skip the header row with the labels "delete," "strip tags," "block," and so forth naming the tasks to perform) and use those to create the `match` attribute value in the generated stylesheet.

The creation of the rest of the stylesheet is very similar, with the element names in question being pulled from the remaining columns of the spreadsheet. For example, to create the template rule that strips elements' tags, the generating stylesheet goes down the spreadsheet's second column grabbing the `table:table-row/table:table-cell[2]/text:p` cell from each row and using these to build a match condition in a template rule like this one:

```
<xsl:template match="para/actor">
  <xsl:apply-templates/>
</xsl:template>
```

(Note that "para/actor" was the only entry in the spreadsheet's second column; additional ones would be included as a pipe-delimited list, as with the example that deletes `IMDB-id` and `crew` elements.) Columns H and I of the spreadsheet, where the user specifies elements that should be output as their own block with a header, require two new techniques to generate the appropriate template rules. First, the generating stylesheet is grabbing two pieces of information from the spreadsheet XML, not one, because each row names the element that gets this treatment at `table:table-row/table:table-cell[8]/text:p` and the header to put above it at `table:table-row/table:table-cell[9]/text:p`. Secondly, unlike the parts of the stylesheet that handle deletion or tag stripping, a single template rule won't handle all of the elements named in column 8, so the generating stylesheet must create a separate template rule for each one in the generated stylesheet.

The most difficult aspect of developing a stylesheet that creates other stylesheets is the XSLT technique required to play the right namespace games. The generating stylesheet must include `xsl:template` elements that are not instructions to the XSLT processor about how to process the source document, but instead are elements to be added to the result document--the generated stylesheet.

The `xsl:namespace-alias` instruction makes this possible. If the `template` element to be added to the output has a prefix representing a dummy URI (for example, if it's named `d:template` and the generating stylesheet maps the "d" prefix to the URI http://some.dummy.placeholder), you can use the `xsl:namespace-alias` instruction to indicate that in the generated stylesheet, elements with the prefix "d" really belong to the http://www.w3.org/1999/XSL/Transform namespace. This way, an XSLT processor running the generated stylesheet would know that `d:template` is actually a proper XSLT instruction.

# 4. Creating both HTML and PDF

The logic necessary for the generating stylesheet to get what it needs from the XML version of the spreadsheet is identical whether it's generating stylesheets that create HTML from the indicated formatting instructions or XSL-FO from those same instructions. Instead of writing two parallel stylesheets that required me to maintain stylesheet logic changes in two places, I set up the generating stylesheet to write a single stylesheet that calls named template rules to output HTML or XSL-FO elements. A stylesheet full of named template rules for HTML creation then uses `xsl:include` to incorporate the stylesheet with the program logic, and a stylesheet full of named template rules for XSL-FO creation includes the same program logic stylesheet. So, for example, to create the template rule that will convert elements to inline italic, when outputting the `xsl:template` element that lists the elements to convert in a pipe-delimited list in its `match` condition, the generating stylesheet calls a template named "outputItalicElement" to insert the generated stylesheet's logic for creating italicized text. When creating HTML, I use this "outputItalicElement" named template:

```
<xsl:template name="outputItalicElement">
  <i><wh:apply-templates/></i>
</xsl:template>
```

(The "wh" prefix in the generating stylesheet is the dummy one used for the namespace trick mentioned earlier. In the generated stylesheet, it will point to the http://www.w3.org/1999/XSL/Transform namespace so that the `apply-templates` element works properly in the generated stylesheet.) When creating XSL-FO output, I use this "outputItalicElement" named template instead:

```
<xsl:template name="outputItalicElement">
  <fo:inline font-style="italic">
    <wh:apply-templates/>
  </fo:inline>
</xsl:template>
```

# 5. Generating stylesheets from Excel spreadsheet XML

The following shows an excerpt from a comparable spreadsheet saved as XML from Microsoft Office Excel 2003:

```
<Table ss:ExpandedColumnCount="256" ss:ExpandedRowCount="97"
       x:FullColumns="1" x:FullRows="1">
  <!-- Column elements omitted -->
 <Row>
  <!-- ss:type="String" attributes omitted from each Data element -->
  <Cell ss:StyleID="s21"><Data>delete</Data></Cell>
  <Cell ss:StyleID="s22"><Data>strip tags</Data></Cell>
  <!-- Row's remaining Cell elements omitted -->
 </Row>
 <Row>
  <Cell ss:StyleID="s25"><Data>IMDB-id</Data></Cell>
  <Cell ss:StyleID="s26"><Data>para/actor</Data></Cell>
```

Modifying the generating stylesheet to read the Excel XML instead of the OpenOffice `content.xml` file only required tweaking the various XPath expressions that pull values out. The logic was still the same: starting at cell A2, go down column A until finding a blank, adding each value found along the way to a pipe-delimited list in the `match` attribute of the generated template rule that deletes elements, and then go through the other columns to pull out the information needed to generate the stylesheet.

Processing the Excel version was actually a bit more straightforward than the OpenOffice version, for two reasons:

- The OpenOffice XML included a doctype declaration that pointed to a DTD. Because my system didn't need the DTD (and the XSLT processor generated an error message when it didn't find it), I used a short perl script to remove the doctype declaration before running the generating stylesheet on the spreadsheet XML.

- The OpenOffice spreadsheet XML reduces the spreadsheet file size by treating the spreadsheet as a sparse matrix when it can. For example, instead of using three empty `table:table-cell` elements to represent the blank cells from column D to column F in row 3, it might have a single empty `table:table-cell` element with a `number-columns-repeated` attribute set to 3. To simplify the generating stylesheet's job when reading OpenOffice spreadsheet XML, I wrote a short `expandcontent.xsl` stylesheet that replaced each `table:table-cell` element that had a `number-columns-repeated` attribute setting with the appropriate number of empty `table:table-cell` elements.

# 6. Taking it home

The spreadsheet you use must be customized to what your shop needs. Instead of a spreadsheet with presentation-oriented formatting instructions like the one shown above, a spreadsheet offering more generic XML-to-XML transformations might offer these tasks instead:

- strip tags from an element
- delete an element or attribute
- rename an element or attribute
- copy an element or attribute verbatim
- reorder an element's children
- convert attribute to element
- specialized processing

You'll recognized a few of these from the spreadsheet above, such as "delete" and "strip tags." Mixing and matching is part of the customization you'll do to create your own spreadsheet template. The main work, of course, is writing the stylesheet that converts your spreadsheet to the production stylesheets that actually strip tags, delete elements, and so forth in your production XML. The zip file [http://www.snee.com/xml/xml2005/xls2xsl.zip] mentioned earlier includes stylesheets named `sxc2xsl.xsl` and `xls2xsl.xsl` can provide a model.

Perhaps the greatest thing about XML, especially in combination with XSLT, is that data stored in this format can easily be converted for other uses. Ever since Microsoft Office tools could store their work in HTML, XML developers have jumped through twisted, fiery hoops trying to convert that HTML into usable XHTML. Office's new ability to save data in proper XML (probably inspired by the example of OpenOffice) not only opens up huge amounts of data to use in XML applications; it also makes it easier to incorporate these tools into XML workflows. While many complain about the XML-based syntax of XSLT, the fact that it is XML makes it easier to automate the generation of XSLT "programs." Put this all together and you can have a pipeline with a spreadsheet program at the beginning and a useful production stylesheet at the end, getting even greater benefit from tools we've all been using for years.

# Bibliography

[XFORMS] *XForms 1.0*, 14 October 2003. Available at http://www.w3.org/TR/2003/REC-xforms-20031014/.

# Biography

Bob **DuCharme**
consulting software engineer
LexisNexis [http://www.lexisnexis.com/]
Charlottesville
Virginia
United States of America
http://www.snee.com/bob

Bob DuCharme is the author of Manning Publications' "XSLT Quickly," Prentice Hall's "XML: The Annotated Specification" and "SGML CD," and McGraw Hill's "Operating Systems Handbook." He writes the monthly "Transforming XML" column for XML.com and has contributed to Dr. Dobb's Journal, perl.com, XML Magazine, XML Journal, IBM developerWorks, XML Developer, O'Reilly Books' "XML Hacks," and Prentice Hall's "XML Handbook." A consulting software engineer at LexisNexis, Bob received his BA in religion from Columbia University and his masters in computer science from New York University. Bob's O'Reilly Network weblog [http://www.oreillynet.com/pub/au/1191] is dedicated to linking-related topics.