# 3. Customizing Content MathML



## 3.1 General Discussion

We can view Content MathML and XSLT (along with an XSLT processor) as an *authoring system* for Presentation MathML. The authoring procedure is the same basic *write, compile, display* model that mathematicians are familar with in LaTeX. If one is using Firefox, the processing is dynamic, so that only the source document and XSLT stylesheet need be published. To create a document that will display with browsers that only support Presentation MathML, the author needs to process the source document (with Content MathML) locally using an external XSLT processor and then publish the output document (with Presentation MathML) on the web. (Recall that the plug-in XSL Results can be used to capture the results of the internal Firefox XLST processor.) Finally, to create a document accessible to virtually anyone, the author could display the source document (with Content MathML) locally with Firefox and then use a Print-to-PDF application to produce a PDF version of the document, which could then be published on the web. Print-to-PDF is a standard feature on Firefox for Linux and for Mac, and is avialable as a plug-in (PrintPDF) for Firefox on Windows.

The three versions of this article (Content MathML, Presentation MathML, PDF) were written in precisely the manner described in the previous paragraph, with the source document (the Content MathML version) written with the simple gedit text editor. Once the Content MathML version of a page has been written, creating the other two version takes mere seconds.

As noted earlier, one could create a completely new XML application for mathematics by writing the appropriate XSLT stylesheet for transforming the new language into Presentation MathML. In general, this would not be a wise approach, since Content MathML is a standard (albeit an incomplete one), and would be a difficult task, since XSLT is a complicated language. A much better approach is to customize and extend standard Content MathML by modifying the existing XSLT stylesheet. In computer programming generally, it is almost always easier to modify an existing, working program than to write a new one from scratch. This general principle is certainly applicable to XSLT.

In the article Math Authoring for the Web Made Easier, XHTML is extended by adding markup for mathematical document structures such as *abstract*, *theorem*, *proof*, *definition* and many others. The customized XHTML is transformed into standard XTHML via an XSLT called remap.xsl. In addition, this article describes a LaTeX-like markup language for mathematics (Content Psuedo-TeX) that is translated ino standard Content MathML. Thus, this article describes a much more ambitious project than ours, but our extension of Content MathML is very similar in spirit to this extension of XHTML.

Finally, it is interesting to note that the general model that I am describing has been implemented several times in non-XML settings. For example, jsMath (and its successor MathJax) uses a LaTeX-like language in the source document which is then tranformed dynamically via JavaScript into standard HTML, styled with CSS, or into Presentation MathML. Similarly, ASCIIMathML uses a simple LaTeX-like markup language which is then transformed dynamically via JavaScript into Presentation MathML. However, I believe that there are distinct benefits to using XML throughout, the approach promoted in this article. In our model, all languages belong to the same XML family that is becoming **the** standard language for the web. All of the XML applications can be processed using open, standard, and robust tools.

## 3.2 Examples

The examples in this subsection explain how certain templates in the standard ctop.xsl stylesheet were modifed for the stylesheet stat.xsl used in my project Virtual Laboratories in Probability and Statistics. A complete summary of the modifications and extensions is given in Section 4.

The easiest extension to Content MathML is the addition of a new special function, used with the standard <apply></apply> construction. One only has to define the name and notation of the special function in the XSLT stylesheet. The standard template for <apply></apply> takes care of everything else.

**Probability**

For my probability project, I wanted special markup for the probability function, denoted with a special double-struck P. Here is the template that I added:

```
<xsl:template mode="c2p" match="mml:prob">
  <mml:mi>&#x02119;</mml:mi>
</xsl:template>
```

The string #x02119; is the unicode designation of a double-struck P (all special entities in XML must be enclosed between the symbols & and ;). With this new template, the markup <prob /> simply results in ℙ while the markup

```
<apply><prob />
  <apply><intersect />
    <ci>A</ci>
    <ci>B</ci>
  </apply>
</apply>
```

for example, results in $\mathbb{P}(A \cap B)$. As you might guess, the probability function appears hundreds of times in my project. Later on, if I want to denote the probability function by Pr, for example, only one trivial change must be made in the stylesheet--replacing ℙ with Pr.

---

My probability project has over 10 new functions, each defined in the same general way. Here is another example that deals with modifying an existing Content MathML construction.

**Cardinality**

Personally, I hate the overuse of vertical bars for such functions as cardinality and determinant, in addition to their basic use in absolute value. The template for cardinality in the standard ctop.xsl stylesheet uses the vertical bars notation:

```
<xsl:template mode="c2p" match="mml:apply[*[1][self::mml:card]]">
  <mml:mrow>
    <mml:mo>|</mml:mo>
    <xsl:apply-templates mode="c2p" select="*[2]"/>
    <mml:mo>|</mml:mo>
  </mml:mrow>
```

```
</xsl:template>
```

Here's how the template works: Line 1 instructs the processor to search the source tree of a given Content MathML expression for an `<apply>` tag whose first child is `<card />`. When it finds a match, the rest of the template brackets the second child with the Presentation MathML markup `<mrow><mo>|</mo>` on the left and `<mo>|</mo></mrow>` on the right. Of course, the templates must be applied to this second child as well, to convert it into Presentation MathML; this is the function of the `apply-templates` part. After all, the second child could itself be a mathematical expression of arbitrary complexity.

But I just want to use function notation for cardinality, with `#` as the function name. Thus, I simply replaced the template above with

```
<xsl:template mode="c2p" match="mml:card">
      <mml:mi>#</mml:mi>
</xsl:template>
```

Now, for example, the markup

```
<apply><card />
  <apply><union />
    <ci>A</ci>
    <ci>B</ci>
  </apply>
</apply>
```

results in #(*A*∪*B*).

---

Introducing a new operator is a bit more work, but is still relatively easy if the new operator works in a manner that similar to a standard operator. One only has to create a copy of the template for the standard operator, and then modify it appropriately. But before we look at our next example, we need to know a bit more about the standard stylesheet ctop.xsl. This stylesheet has three general templates:

- *set*, for set-like constructions in Content MathML such as `<set></set>` and `<list></list>`
- *binary*, for binary operators in Content MathML such as `<apply><minus /></apply>`, `<apply><divide /></apply>`, and `<apply><implies /></apply>`
- *infix*, for (associative) *n*-arry operators such as `<apply><plus /></apply>` and `<apply><times /></apply>`

**Times**

The template in ctop.xsl for the multiplication operation `<apply><times /></apply>` is

```
<xsl:template mode="c2p" match="mml:apply[*[1][self::mml:times]]">
  <xsl:param name="p" select="0"/>
  <xsl:call-template name="infix">
    <xsl:with-param name="this-p" select="4"/>
    <xsl:with-param name="p" select="$p"/>
```

```
    <xsl:with-param name="mo"><mml:mo>&#8290;</mml:mo></xsl:with-param>
  </xsl:call-template>
</xsl:template>
```

Here's how the template works: Line 1 instructs the processor to search the source tree of a given Content MathML expression for an <apply> tag whose first child is <times />. When a match is found, the infix template is called and is passed three argments. Two of the arguments, p and this-p have to do with operator precedence, so that parentheses can be inserted as needed. The third argument is the symbol for the operator, in this case ⁢ which is unicode for "invisible times".

---

We may not completely understand how this template works, but without even looking at the infix template, we can easily create a template for a new operator that works like ordinary multiplication.

**Convolution**

For my probability project, I wanted to create a template for the markup <apply><convolve /></apply> for the convolution of two or more functions. Notationally, this operator works like ordinary multiplication, but with the operator symbol * (and of course applied to functions rather than numbers). Here is the template:

```
<xsl:template mode="c2p" match="mml:apply[*[1][self::mml:convolve]]">
  <xsl:param name="p" select="0"/>
  <xsl:call-template name="infix">
    <xsl:with-param name="this-p" select="4"/>
    <xsl:with-param name="p" select="$p"/>
    <xsl:with-param name="mo"><mml:mo>*</mml:mo></xsl:with-param>
  </xsl:call-template>
</xsl:template>
```

For example, the markup

```
<apply><eq />
  <apply><convolve />
    <ci>f</ci>
   <apply><plus />
      <ci>g</ci>
      <ci>h</ci>
    </apply>>
  </apply>
  <apply><plus />
    <apply><convolve />
      <ci>f</ci>
      <ci>g</ci>
    </apply>
    <apply><convolve />
      <ci>f</ci>
      <ci>h</ci>
    </apply>
```

```
  </apply>
</apply>
```

results in $f*(g+h)=f*g+f*h$. Note that the parentheses on the left have been inserted automatically.

---

Here is another example that involves a simple modification of an existing template.

**Power**

The markup below is the transformation in ctop.xsl that transforms the <apply><power /></apply> construction in Content MathML to Presentation MathML.

```
<xsl:template mode="c2p" match="mml:apply[*[1][self::mml:power]]">
  <mml:msup>
    <xsl:apply-templates mode="c2p" select="*[2]">
      <xsl:with-param name="p" select="5"/>
    </xsl:apply-templates>
    <xsl:apply-templates mode="c2p" select="*[3]">
      <xsl:with-param name="p" select="5"/>
    </xsl:apply-templates>
  </mml:msup>
</xsl:template>
```

Here's how the template works: As before, line 1 instructs the processor to search the source tree of a given Content MathML expression for an <apply> tag whose first child is <power />. When it finds a match, lines 2 and 9 create the <msup></msup> construction from Presentation MathML in the output document. Lines 3 and 6 make the second and third children of our matching node (in the source document) the base and exponent, respectively (in the output document). Of course, the templates must be applied to these children as well, to convert them into Presentation MathML; this is the function of the apply-templates part. Once again, the base and exponent could themselves be mathematical expressions of arbitrary complexity. Finally, lines 4 and 7 have to do with operator precedence, so that appropriate parenthesis can automatically be inserted.

---

Again, even if you don't completely understand this template, it's easy to create a simple modification.

**Convolution Power**

For my probability project, I wanted to create a template for the markup <apply><convolutionpower /></apply> for *convolution power* (the repeated convolution of a function with itself). Notationally, I wanted convolution power to look like an ordinary power, except with the symbol * in the superscript, before the exponent. I did this by making some simple modifications to lines 1 and 6 in power template to create the following template:

```
<xsl:template mode="c2p" match="mml:apply[*[1][self::mml:convolutionpower]]">
  <mml:msup>
    <xsl:apply-templates mode="c2p" select="*[2]">
      <xsl:with-param name="p" select="5"/>
```

```
      </xsl:apply-templates>
      <mml:mrow>
        <mml:mo>*</mml:mo>
        <xsl:apply-templates mode="c2p" select="*[3]">
      </mml:mrow>
        <xsl:with-param name="p" select="5"/>
      </xsl:apply-templates>
    </mml:msup>
</xsl:template>
```

For example, the markup

```
<apply><convolutionpower />
  <apply><plus />
    <ci>f</ci>
    <ci>g</ci>
  </apply>
  <ci>n</ci>
</apply>
```

results in $(f+g)^{*n}$

---

**Binomial Coefficeints**

Our next example considers binomial coefficients (and more generally, multinomial coefficients), essential constructs for probability and combinatorics that are conspicuously missing in standard Content MathML. The new template that was added for the markup `<apply><binomial /></apply>` was derived from the `divide` template in ctop.xsl, and is given below:

```
<xsl:template mode="c2p" match="mml:apply[*[1][self::mml:binomial]]">
    <mml:mrow>
    <mml:mo>(</mml:mo>
    <mml:mfrac linethickness='0'>
      <mml:mrow><xsl:apply-templates mode="c2p" select="*[2]"/></mml:mrow>
      <mml:mrow>
        <xsl:for-each select="*">
          <xsl:if test="position() > 2">
            <xsl:apply-templates mode="c2p" select="."/>
            <xsl:if test="position() != last()">
              <mml:mo>,</mml:mo>
            </xsl:if>
          </xsl:if>
        </xsl:for-each>
      </mml:mrow>
    </mml:mfrac>
    <mml:mo>)</mml:mo>
    </mml:mrow>
```

```
</xsl:template>
```

By now, you may be able to understand the basic sense of the template. First, a left parenthesis is inserted. Next, the Presentation MathML construction for a fraction is opened, with the `linethickness` attribute set to 0. Next, the second child of `<apply>` is put in the numerator position (the first child is `<binomial />`), while subsequent children are put in the denominator position, separated by commas. Finally, the fraction construction is closed and the right parenthesis inserted. For example, the markup

```
<apply><eq />
  <apply><binomial />
    <ci>n</ci>
    <ci>k</ci>
  </apply>
  <apply><binomial />
    <ci>n</ci>
    <apply><minus />
      <ci>n</ci>
      <ci>k</ci>
    </apply>
  </apply>
</apply>
```

gives $\binom{n}{k} = \binom{n}{n-k}$

---

Our last example is more specialized and illustrates the use of an attribute.

**Hypothesis testing**

Hypothesis testing is a core technique in statistics, and uses some specialized notation. For my project, I wanted to use the markup `<hypothesis />` with the standard `<apply></apply>` construction, and with a `type` attribute to distinguish between a null and an alternative hypothesis. Here are the templates:

```
<xsl:template mode="c2p" match="mml:apply[*[1][self::mml:hypothesis[@type='null']]]">
  <mml:mrow>
    <mml:msub>
      <mml:mi>H</mml:mi>
      <mml:mn>0</mml:mn>
    </mml:msub>
    <mo>:  </mo>
    <xsl:apply-templates mode="c2p" select="*[2]"/>
  </mml:mrow>
</xsl:template>

<xsl:template mode="c2p" match="mml:apply[*[1][self::mml:hypothesis[@type='alt']]]">
  <mml:mrow>
    <mml:msub>
      <mml:mi>H</mml:mi>
```

```
      <mml:mn>1</mml:mn>
    </mml:msub>
    <mo>:  </mo>
    <xsl:apply-templates mode="c2p" select="*[2]"/>
  </mml:mrow>
</xsl:template>
```

These templates are fairly easy to undersand. The symbol *H* in given either the subscript 0 (for a null hypothesis) or 1 (for an alternative hypothesis), depending on the type attribute. This symbol is followed by the actual hypothesis (the second child), and the entire construction is wrapped in <mrow></mrow>. For example, the markup

```
<apply><hypothesis type="alt" />
  <apply><neq />
    <ci>&mu;</ci>
    <cn>0</cn>
  </apply>
</apply>
```

results in $H_1$: $\mu \neq 0$

---

To summarize, there are two main advantages that I have gained by using specialized Content MathML. First, I am able to have more complete separation of content from presentation. If I change my mind about notation, I can make simple, sometimes trivial modifications in the stylesheet that will cause the notational changes in every page of my project. Second, the mathematics in the source document is much easier to read and understand (after a bit of practice). The next section gives a complete summary of my customized and extended version of Content MathML.

---