Content-Faithful Stylesheets for MathML

Igor Rodionov   Stephen M. Watt

Ontario Research Centre for Computer Algebra
Department of Computer Science
University of Western Ontario
London Ontario, Canada

{igor,watt}@csd.uwo.ca

ORCCA Technical Report TR-00-24

Abstract

XSLT stylesheets may be used to transform MathML expressions, for example to render Content MathML to Presentation MathML. It is desirable, however, that XSLT stylesheets for MathML preserve the original meaning of the expressions. These are called *content-faithful* stylesheets. This paper presents different styles of content-faithful transformation for MathML, preserving the original Content MathML in the output Presentation MathML in different ways. These methods can be used to make existing transformations content-faithful with minimal alteration of the original stylesheet.

1 Introduction

The MathML 1 specification [1] provides a mechanism for mathematical expressions to be given with an XML syntax. This is intended to allow a more natural inclusion of mathematics in web pages and digital libraries, as well as providing a format for the exchange of mathematical data between programs. The use of XML syntax allows mathematics to participate as first-class content, on par with text, so that it can be treated with all the same tools. One such tool is XSLT[3], a language for transforming XML documents into other XML documents.

MathML provides two sets of markup elements, those that specify the appearance, or *presentation*, of a mathematical expression and those that specify the semantics, or *content*, of a mathematical expression. In addition, there are certain elements that apply to both sorts of markup, including the `<semantics>` element that allows a MathML expression to have text or XML annotations. A single expression may be given with presentation markup, content markup, or a combination. An expression containing only presentation markup is said to be a fragment of *Presentation MathML*. Likewise an expression containing only content markup is said to be a fragment of *Content MathML*.

One of the authors of the present report has been involved in the definition of MathML 1 and MathML 2, and has contributed toward the standard's handling of mixed content and presentation markup. The MathML 1 specification contemplates both presentation markup contained in content markup and content markup contained in presentation markup. The MathML2 Candidate Recommendation [2] goes further and elaborates the possibility of using the MathML `<semantics>` element to annotate a MathML expression with another MathML expression. The particular case of providing Presentation MathML with a corresponding Content MathML annotation, or vice versa, is described as *parallel markup*. The Candidate Recommendation contemplates both top-level and fine-grained parallel markup.

In the MathML 2 Candidate Recommendation we observe that notational stylesheets may be used to provide a specific notation for the display of Content MathML. We also note that, in the implementation of such stylesheets, there will be a temptation to produce Presentation MathML output that loses track of its Content MathML origin.

Under many circumstances, such transformations that lose track of the original expression semantics will be undesirable. The simplest example is when an expression is ambiguous: Does the expression J_α refer to a Bessel function or to an angular momentum vector? We do not know, unless the original Content MathML is retained. A second example is given by the scenario of a mathematical display editor that displays mathematical expressions according to some notational style sheet. When a subexpression of the displayed presentation expression is selected, it is necessary to find the corresponding content subexpression.

To deal with situations such as this, we define a *content-faithful* transformation as one that retains the original content in parallel markup. In the MathML 2 Candidate Recommendation, we suggest that tools that support MathML should be content-faithful when possible.

This article shows techniques to write content-faithful XSLT stylesheets. We take a simple example and give a detailed explanation of three different approaches to content-faithful transformation. These approaches may be applied after the fact to existing stylesheets to make them content-faithful while requiring only minimal modifications to the original code.

2 A Simple Transformation Problem

2.1 Definition

To be as clear as possible, we work with a simple transformation problem. This will keep the examples from being cluttered with too many details. We will use this example to illustrate general techniques to convert a presentation-only transformation to various forms of content-faithful transformations.

For the illustrative examples we restrict our attention to transformations that accept Content MathML satisfying the following grammar:

$$\begin{array}{rcl}
CExpr & ::= & CLeaf \\
 & \mid & \texttt{<apply>}\ COp\ \ CExpr\ \ CExpr\ \texttt{</apply>} \\
 & & \\
CLeaf & ::= & \texttt{<cn>}\ PCDATA\ \texttt{</cn>} \\
 & \mid & \texttt{<ci>}\ PCDATA\ \texttt{</ci>} \\
 & & \\
COp & ::= & \texttt{<plus/>} \\
 & \mid & \texttt{<times/>}
\end{array}$$

The generated Presentation MathML will use the grammar:

$$\begin{array}{rcl}
PExpr & ::= & PLeaf \\
 & \mid & \texttt{<mfenced>}\ PExpr\ \ POp\ \ PExpr\ \texttt{</mfenced>} \\
 & & \\
PLeaf & ::= & \texttt{<mn>}\ PCDATA\ \texttt{</mn>} \\
 & \mid & \texttt{<mi>}\ PCDATA\ \texttt{</mi>} \\
 & & \\
POp & ::= & \texttt{<mo>+</mo>} \\
 & \mid & \texttt{<mo>*</mo>}
\end{array}$$

That is, we have an expression made up of nested binary sums and products of numbers and identifiers, and produce a fully parenthesized presentation using infix operators. A more realistic example, still restricted to sums and products, would allow n-ary operations and only use parentheses when necessary in the output.

We wish to transform the expression in the obvious way, i.e.

$$\begin{array}{rcl}
\texttt{<cn>}\alpha\texttt{</cn>} & \Rightarrow & \texttt{<mn>}\alpha\texttt{</mn>} \\
\texttt{<ci>}\alpha\texttt{</ci>} & \Rightarrow & \texttt{<mi>}\alpha\texttt{</mi>} \\
\texttt{<plus/>} & \Rightarrow & \texttt{<mo>+</mo>} \\
\texttt{<times/>} & \Rightarrow & \texttt{<mo>*</mo>} \\
\texttt{<apply>}\ \mu\ \alpha\ \beta\ \texttt{</apply>} & \Rightarrow & \texttt{<mfenced>}\ \tau(\alpha)\ \tau(\mu)\ \tau(\beta)\ \texttt{</mfenced>}
\end{array}$$

where $\tau(\alpha)$ is the result of applying the transformation recursively to α.

2.2 Example

We take as input the Content MathML expression

```
<apply>
   <times/>
   <apply> <plus/> <ci>a</ci> <cn>1</cn> </apply>
   <apply> <plus/> <ci>b</ci> <cn>2</cn> </apply>
</apply>
```

This should be transformed to the Presentation MathML expression

```
<mfenced>
   <mfenced> <mi>a</mi> <mo>+</mo> <mn>1</mn> </mfenced>
   <mo>*</mo>
   <mfenced> <mi>b</mi> <mo>+</mo> <mn>2</mn> </mfenced>
</mfenced>
```

2.3 Stylesheet

The following is an XSLT stylesheet that implements the transformation specified in Section 2.1:

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

   <xsl:template match="plus" > <mo>+</mo> </xsl:template>
   <xsl:template match="times"> <mo>*</mo> </xsl:template>

   <xsl:template match="cn">
      <mn><xsl:copy-of select="text()"/></mn>
   </xsl:template>
   <xsl:template match="ci">
      <mi><xsl:copy-of select="text()"/></mi>
   </xsl:template>

   <xsl:template match="apply">
      <mfenced>
          <xsl:apply-templates select="*[2]"/>
          <xsl:apply-templates select="*[1]"/>
          <xsl:apply-templates select="*[3]"/>
      </mfenced>
   </xsl:template>
</xsl:stylesheet>
```

As well as handling additional elements, a full converter would also need to have rules to copy attributes, insert parentheses where needed, etc.

3 Three Content-Faithful Styles

There are three main possibilities for the output of a content-faithful transformation:

1. top-level parallel markup
2. fine-grained parallel markup with recursion
3. fine-grained parallel markup with cross references.

We show the output that each of these would give for our example.

3.1 Top-Level Parallel Markup

The first of these, top-level parallel markup, would give the following output for the example of Section 2.2.

```
<semantics>
   <mfenced>
      <mfenced> <mi>a</mi> <mo>+</mo> <mn>1</mn> </mfenced>
      <mo>*</mo>
      <mfenced> <mi>b</mi> <mo>+</mo> <mn>2</mn> </mfenced>
   </mfenced>
   <annotation-xml encoding="MathML-Content">
      <apply>
         <times/>
         <apply> <plus/> <ci>a</ci> <cn>1</cn> </apply>
         <apply> <plus/> <ci>b</ci> <cn>2</cn> </apply>
      </apply>
   </annotation-xml>
</semantics>
```

This provides the original Content MathML for the whole expression, but not for the subexpressions. For that, one must use one of the fine-grained parallel markup alternatives.

3.2 Fine-Grained Parallel Markup with Recursion

For fine-grained parallel markup with recursion, each Presentation node is annotated with the corresponding Content MathML. For the example of Section 2.2, the fine-grained parallel markup with recursion would be as follows:

```
<semantics>
   <mfenced>
      <semantics>
         <mfenced>
            <semantics>
               <mi>a</mi>
               <annotation-xml encoding="MathML-Content"> <ci>a</ci> </annotation-xml>
            </semantics>
            <semantics>
               <mo>+</mo>
               <annotation-xml encoding="MathML-Content"> <plus/> </annotation-xml>
            </semantics>
            <semantics>
               <mn>1</mn>
               <annotation-xml encoding="MathML-Content"> <cn>1</cn> </annotation-xml>
            </semantics>
         </mfenced>
         <annotation-xml encoding="MathML-Content">
            <apply> <plus/> <ci>a</ci> <cn>1</cn> </apply>
         </annotation-xml>
      </semantics>
      <semantics>
         <mo>*</mo>
         <annotation-xml encoding="MathML-Content"> <times/> </annotation-xml>
      </semantics>
      <semantics>
         <mfenced>
            <semantics>
               <mi>b</mi>
               <annotation-xml encoding="MathML-Content"> <ci>b</ci> </annotation-xml>
            </semantics>
            <semantics>
               <mo>+</mo>
               <annotation-xml encoding="MathML-Content"> <plus/> </annotation-xml>
            </semantics>
            <semantics>
               <mn>2</mn>
               <annotation-xml encoding="MathML-Content"> <cn>2</cn> </annotation-xml>
            </semantics>
         </mfenced>
         <annotation-xml encoding="MathML-Content">
            <apply> <plus/> <ci>b</ci> <cn>2</cn> </apply>
         </annotation-xml>
      </semantics>
   </mfenced>
   <annotation-xml encoding="MathML-Content">
      <apply>
         <times/>
         <apply> <plus/> <ci>a</ci> <cn>1</cn> </apply>
         <apply> <plus/> <ci>b</ci> <cn>2</cn> </apply>
      </apply>
   </annotation-xml>
</semantics>
```

This method involves repetition, for example the Content MathML for all leaf expressions is repeated at each level from the leaf to the root. To avoid this repetition, we use fine-grained parallel markup with cross references.

3.3 Fine-Grained Parallel Markup with Cross References

Fine-grained parallel markup with cross references uses only one instance of the Content MathML expression. Its sub-parts are connected to the Presentation MathML subexpressions via cross references. For our example of Section 2.2, the markup would be:

```
<semantics>
   <mfenced xref="L0">
      <mfenced xref="L2">
         <mi xref="L4">a</mi> <mo xref="L3">+</mo> <mn xref="L5">1</mn>
      </mfenced>
      <mo xref="L1">*</mo>
      <mfenced xref="L6">
         <mi xref="L8">b</mi> <mo xref="L7">+</mo> <mn xref="L9">2</mn>
      </mfenced>
   </mfenced>
   <annotation-xml encoding="MathML-Content">
      <apply id="L0">
         <times id="L1"/>
         <apply id="L2"> <plus id="L3"/> <ci id="L4">a</ci> <cn id="L5">1</cn> </apply>
         <apply id="L6"> <plus id="L7"/> <ci id="L8">b</ci> <cn id="L9">2</cn> </apply>
      </apply>
   </annotation-xml>
</semantics>
```

Note that the `id` attributes are placed on the annotating Content MathML, and not on the annotated Presentation MathML. This is the opposite of the example in the Candidate Recommendation. Both are legitimate MathML, but the way we have shown here is more natural for output generated from the Content MathML expression.

This form of fine-grained parallel markup can assume that the input already has `id` attributes on the elements of interest, and generate `xref` attributes on the corresponding generated Present elements. Alternatively, if it is desired to cross-link all nodes, a first pass stylesheet can place generated `id` attributes on elements that do not have an `id` already.

4 Stylesheets for the Three Approaches

We show modified stylesheets that implement content-faithful versions of the transformation for our simple example. Each section shows the stylesheet for one of the content-faithful transformation styles.

4.1 Top-Level Parallel Markup

We can obtain top-level parallel markup by adding one template for the root node to our original style sheet of Section 2.3". The new template, specified by `match="/"`, generates a `<semantics>` element with the two children. The first child is the result of applying the original Content to Presentation transformations to the Content MathML input. The second child is an `<annotation-xml>` element containing a copy of the original Content MathML.

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

   <!-- New rule -->

   <xsl:template match="/">
      <semantics>
         <xsl:apply-templates/>
         <annotation-xml encoding="MathML-Content">
            <xsl:copy-of select="."/>
         </annotation-xml>
      </semantics>
   </xsl:template>

   <!-- Old rules -->

   <xsl:template match="plus" > <mo>+</mo> </xsl:template>
   <xsl:template match="times"> <mo>*</mo> </xsl:template>

   <xsl:template match="cn">
      <mn><xsl:copy-of select="text()"/></mn>
   </xsl:template>
   <xsl:template match="ci">
      <mi><xsl:copy-of select="text()"/></mi>
   </xsl:template>

   <xsl:template match="apply">
      <mfenced>
          <xsl:apply-templates select="*[2]"/>
          <xsl:apply-templates select="*[1]"/>
          <xsl:apply-templates select="*[3]"/>
      </mfenced>
   </xsl:template>
</xsl:stylesheet>
```

4.2 Fine-Grained Parallel Markup with Recursion

Fine-grained parallel markup with recursion may be obtained simply using XSLT modes. We do this by adding two templates to the original stylesheet and defining two modes: `xform` and `semantics`.

The `xform` mode performs the all transformations of the original stylesheet. However, whenever these rules apply transformation templates recursively the do so in `semantics` mode.

The `semantics` mode uses our new rule that matches all elements. This rule does the same transformation at each interior node that the rule for the root node does in top-level parallel markup: It generates a `<semantics>` element with two children. The first child is the result of applying the Content to Presentation transformations to the Content MathML node (in mode `xform`). The second child retains the original Content MathML node in an `<annotation-xml>` element.

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

   <!-- New rules -->

   <xsl:template match="/">
       <xsl:apply-templates select="." mode="semantics"/>
   </xsl:template>

   <xsl:template match="/|*" mode="semantics">
      <semantics>
         <xsl:apply-templates select="." mode="xform"/>
         <annotation-xml encoding="MathML-Content">
            <xsl:copy-of select="."/>
         </annotation-xml>
      </semantics>
   </xsl:template>

   <!-- Old rules, with modes added -->

   <xsl:template match="plus"  mode="xform"> <mo>+</mo> </xsl:template>
   <xsl:template match="times" mode="xform"> <mo>*</mo> </xsl:template>

   <xsl:template match="cn" mode="xform">
      <mn><xsl:copy-of select="text()"/></mn>
   </xsl:template>
   <xsl:template match="ci" mode="xform">
      <mi><xsl:copy-of select="text()"/></mi>
   </xsl:template>

   <xsl:template match="apply" mode="xform">
      <mfenced>
          <xsl:apply-templates select="*[2]" mode="semantics"/>
          <xsl:apply-templates select="*[1]" mode="semantics"/>
          <xsl:apply-templates select="*[3]" mode="semantics"/>
      </mfenced>
   </xsl:template>
</xsl:stylesheet>
```

4.3 Fine-Grained Parallel Markup with Cross-References

For many applications the original Content MathML will not already have `id` attributes on the elements. We therefore first present a stylesheet that may be used to paint generated `id` values on all elements. Secondly, we present a stylesheet that generates fine-grained parallel markup that uses whatever `id` attributes are on the tree. Finally we present a method to simplify the resulting output.

4.3.1 Generating Id Attributes

The following stylesheet generates `id` attributes on all element nodes, and copies through text and attributes.

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

   <xsl:template match="/|*|@*">
      <xsl:copy>
            <xsl:attribute name="id">
               <xsl:value-of select="generate-id()"/>
            </xsl:attribute>
         <xsl:apply-templates select="*|@*|text()"/>
      </xsl:copy>
   </xsl:template>

</xsl:stylesheet>
```

The output of the `generate-id` function will depend on the XSLT implementation. Using the *XT* [4] XSLT processor, this stylesheet gives the following output when applied to our example input.

```
<apply id="N1">
   <times id="N3"/>
   <apply id="N6">  <plus id="N8"/>  <ci id="N10">a</ci> <cn id="N13">1</cn> </apply>
   <apply id="N17"> <plus id="N19"/> <ci id="N21">b</ci> <cn id="N24">2</cn> </apply>
</apply>
```

(In this and the following examples we have hand adjusted the output indentation and spacing for compactness.)

4.3.2 Generating Parallel Markup

The stylesheet below produces fine grained parallel markup with cross references. It uses a top-level semantics element containing the generated Presentation MathML and having the input Content MathML as an annotation. When a node of the input Content MathML has an `id` attribute, a corresponding `xref` is given in the generated Presentation MathML. As we did for recursive parallel markup, we use XSLT modes to preserve the form of the original rule set.

This stylesheet wraps each generated presentation node with an `<mrow>` containing the `xref` attribute. It would be possible to place the `xref` attributes on the generated nodes (`<mfenced>`, `<mi>`, *etc*), but this would involve heavily modify each rule of the original stylesheet and the the logical structure of the original transformation would be lost.

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

   <!-- New rules -->
   <xsl:template match="/">
      <semantics>
         <xsl:apply-templates select="." mode="xref"/>
         <annotation-xml encoding="MathML-Content">
            <xsl:copy-of select="."/>
         </annotation-xml>
      </semantics>
   </xsl:template>

   <xsl:template match="*" mode="xref">
      <xsl:choose>
         <xsl:when test="@id">
            <mrow xref="{@id}">
               <xsl:apply-templates select="." mode="xform"/>
            </mrow>
         </xsl:when>
         <xsl:otherwise>
               <xsl:apply-templates select="." mode="xform"/>
         </xsl:otherwise>
      </xsl:choose>
   </xsl:template>

   <!-- Old rules, with modes added -->
   <xsl:template match="plus"  mode="xform"> <mo>+</mo> </xsl:template>
   <xsl:template match="times" mode="xform"> <mo>*</mo> </xsl:template>

   <xsl:template match="cn" mode="xform">
      <mn><xsl:copy-of select="text()"/></mn>
   </xsl:template>
   <xsl:template match="ci" mode="xform">
      <mi><xsl:copy-of select="text()"/></mi>
   </xsl:template>

   <xsl:template match="apply" mode="xform">
      <mfenced>
          <xsl:apply-templates select="*[2]" mode="xref"/>
          <xsl:apply-templates select="*[1]" mode="xref"/>
          <xsl:apply-templates select="*[3]" mode="xref"/>
      </mfenced>
   </xsl:template>
</xsl:stylesheet>
```

This stylesheet gives the following output for our example.

```
<semantics>
   <mrow xref="N1"> <mfenced>
      <mrow xref="N6"> <mfenced>
         <mrow xref="N10"> <mi>a</mi> </mrow>
         <mrow xref="N8">  <mo>+</mo> </mrow>
         <mrow xref="N13"> <mn>1</mn> </mrow>
      </mfenced> </mrow>
      <mrow xref="N3"> <mo>*</mo> </mrow>
      <mrow xref="N17"> <mfenced>
         <mrow xref="N21"> <mi>b</mi> </mrow>
         <mrow xref="N19"> <mo>+</mo> </mrow>
         <mrow xref="N24"> <mn>2</mn> </mrow>
      </mfenced> </mrow>
   </mfenced> </mrow>

   <annotation-xml encoding="MathML-Content">
      <apply id="N1">
         <times id="N3"/>
         <apply id="N6"> <plus id="N8"/> <ci id="N10">a</ci><cn id="N13">1</cn></apply>
         <apply id="N17"><plus id="N19"/><ci id="N21">b</ci><cn id="N24">2</cn></apply>
      </apply>
   </annotation-xml>
</semantics>
```

4.3.3 Simplified Output

If it is desired to place the `xref` attributes directly on the generated Presentation elements, then this may be achieved with an additional pass.

In order to avoid confusion with any `<mrow>` elements that may have been in the original input, we assume that the transformation stylesheet instead generates some application-specific element to contain the `xref` attributes. Here we denote this element as `<app:xref-holder>`. In this case, the second template of the preceding stylesheet would be replaced with

```
   <xsl:template match="*" mode="xref">
      <xsl:choose>
         <xsl:when test="@id">
            <app:xref-holder xref="{@id}">
               <xsl:apply-templates select="." mode="xform"/>
            </app:xref-holder>
         </xsl:when>
         <xsl:otherwise>
               <xsl:apply-templates select="." mode="xform"/>
         </xsl:otherwise>
      </xsl:choose>
   </xsl:template>
```

With this modification to the transforming stylesheet, we may use the following clean-up stylesheet to move the `xref` attributes to the desired Presentation MathML elements.

```
<xsl:stylesheet version="1.0"
   xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
   xmlns:app="http://www.csd.uwo.ca/~watt/example-application" >

   <xsl:output method="xml" indent="yes"/>

   <xsl:template match="app:xref-holder">
         <xsl:apply-templates select="*">
            <xsl:with-param name="xref_val" select="@xref"/>
         </xsl:apply-templates>
   </xsl:template>

   <xsl:template match="/|*|@*">
      <xsl:param name="xref_val"/>
      <xsl:copy>
         <xsl:if test="$xref_val!=''">
            <xsl:attribute name="xref">
               <xsl:value-of select="$xref_val"/>
            </xsl:attribute>
         </xsl:if>
         <xsl:apply-templates select="*|@*|text()"/>
      </xsl:copy>
   </xsl:template>

</xsl:stylesheet>
```

With this, the output from our example becomes

```
<semantics>
   <mfenced xref="N1">
      <mfenced xref="N6">
         <mi xref="N10">a</mi> <mo xref="N8">+</mo> <mn xref="N13">1</mn>
      </mfenced>
      <mo xref="N3">*</mo>
      <mfenced xref="N17">
         <mi xref="N21">b</mi> <mo xref="N19">+</mo> <mn xref="N24">2</mn>
      </mfenced>
   </mfenced>
   <annotation-xml encoding="MathML-Content">
      <apply id="N1">
         <times id="N3"/>
         <apply id="N6"> <plus id="N8"/> <ci id="N10">a</ci><cn id="N13">1</cn></apply>
         <apply id="N17"><plus id="N19"/><ci id="N21">b</ci><cn id="N24">2</cn></apply>
      </apply>
   </annotation-xml>
</semantics>
```

5 Conclusions

We have shown three simple methods to make XML transformations content-faithful, with one method for each of the possible parallel markup forms. Which methods is best to use depends on the form of parallel markup preferred by the application.

Each of the three methods preserves the structure of the original stylesheet and adds only a few additional templates, perhaps requiring XSLT modes.

We conclude that it is quite easy to make XML transformations content-faithful, and that applications can be content-faithful with very little additional complexity.

References

[1] *Mathematical Markup Language (MathML) 1.01 Specification*,
Patrick Ion, Robert Miner (editors), Stephen Buswell, Stan Devitt, Angel Diaz, Patrick Ion, Robert Miner, Nico Poppelier, Bruce Smith, Neil Soiffer, Robert Sutor, Stephen Watt (authors), W3C Recommendation, revision of 7 July 1999, REC-MathML-19980407,
`http://www.w3.org/1999/07/REC-MathML-19990707`.

[2] *Mathematical Markup Language (MathML) Version 2.0*,
David Carlisle, Patrick Ion, Robert Miner, Nico Poppelier (editors), Ron Ausbrooks, Stephen Buswell, Stéphane Dalmas, Stan Devitt, Angel Diaz, Roger Hunter, Bruce Smith, Neil Soiffer, Robert Sutor, Stephen Watt (authors), W3C Candidate Recommendation 13 November 2000,
`http://www.w3.org/TR/2000/CR-MathML2-20001113`.

[3] *XSL Transformations (XSLT) Version 1.0*,
James Clark (editor), W3C Recommendation 16 November 1999
`http://www.w3.org/TR/1999/REC-xslt-19991116`.

[4] *XT Implementation of XSLT*,
James Clark, `http://www.jclark.com`