

MathML without Plugins using VML

A Writing Project

Presented to

The Faculty of the Department of Computer Science

San Jose State University

In Partial Fulfillment of the Requirement for the Degree

Master of Science

By

Namon Nuttayasakul

May 2003

May 2003

Namon Nuttayasakul
namonpick@hotmail.com

ALL RIGHTS RESERVED

APPROVED FOR THE DEPARTMENT OF COMPUTER SCIENCE

Dr. Christopher Pollett

Dr. Michael Beeson

Dr. Agustin Araya

APPROVED FOR THE UNIVERSITY

Abstract

MathML is an XML based mark-up language for displaying mathematical expressions on the web and there exist programs to convert both TeX and LaTeX to this format. Unfortunately, most popular browsers such as Internet Explorer and Netscape Navigator do not natively support this language. The purpose of this project is to develop a stylesheet transformation to convert MathML file into a viewable format that natively support by popular browsers, while conserving the quality of rendering mathematical expression by using Vector Graphic. This project uses XSLT (eXtensible Stylesheet Language Transformations) to convert XML file that contains MathML to XML file containing HTML and VML (Vector graphic Markup Language). As a result, it takes shorter time to view the mathematical expressions on the Web than the conventional way that uses image files such as GIF or JPEG and also gives a better rendering result than those displaying with plain HTML.

Table of contents

	Page
Introduction	1
Related Works.....	3
Background.....	6
Requirement.....	15
Design.....	17
Implementation.....	27
Deployment.....	61
Conclusion	62
Bibliography.....	63

1) Introduction

The standard way to view math on the web is to write the mathematical document using LaTeX and use a conversion program such as latex2html [D96], WebEQ [D02], TtH [H01], or HeVeA [M02]. The first two programs generate graphic images (WebEQ can output MathML and applet code) for each mathematical formula on the page. This makes such web-documents both slow to load and hard to maintain as they now consist of many files, one for each formula. The latter two programs convert LaTeX directly into a single HTML file and try to draw the equations as best as possible given the limitations of HTML. These programs are very fast and often produce reasonable results. However, picture environments and the like still must be output using graphics images. MathML is an XML based mark-up language for displaying math on the web and there does exist programs to convert both TeX and LaTeX to this format. Unfortunately, neither Netscape Navigator nor Internet Explorer natively supports this language. SVG [W3C01c] and VML [W3C01b] are XML-defined vector mark-up languages in which both math notations and picture environments can easily be rendered. Currently, VML is natively supported in Internet Explorer, but SVG is neither natively supported by Internet Explorer nor Netscape Navigator.

Style-sheet transformations are rules that are applied by the browser or by the server to a tag when it is read or before it is transmitted. They basically provide a mechanism by which a document can be “compiled” into a format displayable by a browser. Netscape Navigator currently supports CSS2 (cascading style sheets level 2) [W3C00b], but with the intention to have built in support for XSLT (eXtensible

Stylesheet Language Transformations) [W3C01d] in the future, the latter being stronger and more flexible. Internet Explorer supports XSLT. However, CSS2 has some support for tag-replacement by hypertext. This allows tags to be replaced by ECMAScript (aka Javascript) code [ECMA99], which in theory can output SVG code.

This project uses the XSLT transformation language to convert XML file that contains MathML to XML file containing HTML and VML. The document only requires a single processing instruction link to the stylesheet. The stylesheet transforms the supplied XML file, adding whatever markup is required to render MathML in the current browser, and passes the resulting document to the browser for rendering. Clearly this does require that the browser supports XSLT transformations, which means that a relatively new browser is required, however the current versions of (at least) Internet Explorer supports XSLT, so while XSLT support is not universal, it is, or soon will be, available on the majority of desktop browsers. Thus, in both Netscape Navigator and Internet Explorer clients, it should be feasible to produce style-sheet transformations from MathML to the target language. Nevertheless, such a translation would be difficult to perform and is worthy of a master.

This document is organized as follows. Section 1 (Introduction) provides information on the related research and background of this project. Section 2 (Requirement) states the goal of this master project in detail. Section 3 (Design) provides information on the architecture, design pattern used for this project. Section 4 (Implementation) covers important implementation issues and algorithms used to accomplish each step of this project. Section 5 (Deployment) presents some comparison experiments. Section 6 (Conclusion) contains conclusions of my master project.

1.1) Related Works

This section will introduce related material as well as the other technologies available to view MathML on the Web. Although, MathML is becoming more and more widely used, few browsers have "native" support for MathML, and only one has native support for the Content part of MathML. Currently, Mozilla version 0.9.9 is the only browser that fully supports MathML in both presentation and content mode. The Amaya browser supports only the presentation mode of MathML. However, popular browsers such as Internet Explorer (IE) or Netscape Navigator still need some plug-ins or a range of extensions (in particular WebEQ and MathPlayer from Design Science and Techexplorer from IBM) in order to make them able to display MathML. Below is the summary list of browsers that will render the pages categorized by the operating system as shown:

Operating System	Browser with necessary configuration
Windows	IE 5.0 with the Techexplorer plug-in
	IE 5.5 with either the MathPlayer or Techexplorer plug-ins
	IE 6.0, optionally with MathPlayer or Techexplorer plug-ins
	Netscape 6.1 with Techexplorer plug-in
	Amaya (Presentation MathML only)
	Mozilla 0.9.9
Macintosh	IE 5.0 with Techexplorer plug-in
	Mozilla 0.9.9
Linux/Unix	Netscape 6.1 with Techexplorer plug-in
	Mozilla 0.9.9
	Amaya (Presentation MathML only)

The disadvantage of using a third party extension to render MathML within a web page is that it requires a specific markup to specify the rendering extension. The use of such a markup ties the document to one particular platform; whereas, the ideal of publishing information on the Web is that it should be accessible to all using a range of tools.

Related work that contributes to make MathML viewable on the Web is a stylesheet [W3C22a] provided by W3C for XHTML file that has the MathML code as an embedded tag. This is a mechanism that uses the XSLT transformation language to allow an XML file that contains MathML to convert to an XML file containing XHTML. This will work in most cases (but not on Internet Explorer: for security reasons IE will not execute an XSLT stylesheet that is not located on the same server as the XHTML+MathML document). However, there are alternatives, such as processing it on the client side. This mechanism basically transforms a MathML document into an XHTML document, which of course will give a less satisfying result as it tries to draw the mathematical expression using XHTML capability. The result will definitely be of lower quality when compared to the ideas proposed by this Master's project, which uses a Vector graphic language to draw each expression. Furthermore, my project will also give pages with better printing quality.

According to W3C, many works on stylesheet transformation has been designed to support a range of MathML renderers. A XSLT stylesheet transformation is chosen to be a candidate for solving a problem of platform-dependant rendering. One of them is to use MathML with an Universal MathML stylesheet. To accomplish this, a MathML author needs to add a link within the MathML file to a universal stylesheet,

which is a stylesheet that detects the type of browser and MathML render, then uses that information to transform and add some necessary markup to enable MathML rendering according to specific browsers.

The main platform's specific modifications that are included in the universal stylesheet are as follows: transformation from MathML content mode to MathML presentation mode. This transformation allows rendering in those browsers that only support presentation MathML, such as Netscape Navigator and Mozilla. Adding an `<object>` element to the document head to specify some Microsoft behavior extension for a fixed namespace prefix transforms the entire document to have this prefix for every MathML element. This is used for Techexplorer and Mathplayer in Internet Explorer. Next, applet and plugin interfaces are used in dealing with older versions of browsers by transforming MathML as a string within an attribute for using with an `<applet>` or `<embed>` element. Lastly, for the default, this universal stylesheets will render MathML using CSS and Javascript.

1.2) Background

In CS297, I have finished five deliverables each of which contribute to the learning process for the final project. The five deliverables are listed as follows:

1. Become reasonably proficient with VML and SVG. This will be demonstrated by creating an image of a sunset in both of these languages.
2. Become reasonably proficient at MathML and LaTeX. This was to be demonstrated by reproducing pages 130 and 131 of A Guide to LaTeX2e in these languages.
3. Create a DTD for drawing matrices. Write a XSL transform to render this language in VML.
4. Get the MathML matrix related, apply, minus, times, divide, and eq tags to translate to VML and to SVG via XSLT.
5. A first semester 10-20 paged report on the project. This report is the result of the fifth deliverable.

The following subsections combine some research I collected from delivering these deliverables during CS297.

1.2.1) Deliverable 1: Become reasonably proficient with VML and SVG. This will be demonstrated by creating an image of a sunset in both of these languages.

The purpose of this deliverable was to learn and experiment with the currently available vector markup languages. I needed to become proficient with some of the

functions and features of these languages in order to be able to render mathematical expressions in the vector graphic for the final project.

Today, the most available, well-known vector markup languages are VML and SVG. According to my research during this semester, there are many advantages of vector graphic format over other image formats, particularly over JPEG, and GIF, the most common graphic formats used on the Web today. The Advantages of vector graphic format over other graphic formats are provided below:

- ***Plain text format:*** Vector graphic files can be read and modified by a range of tools, and are usually much smaller and more compressible than JPEG or GIF image.
- ***Scalable:*** Unlike bitmapped GIF and JPEG formats, vector format images can be printed with high quality at any resolution, without the "staircase" effects you see when printing bitmapped images.
- ***Zoomable:*** Vector graphics allow you to zoom in on any portion of an image and not see any degradation.
- ***Searchable and selectable text:*** Unlike in bitmapped images, text in vector graphic markup language is selectable and searchable. For example, you can search for specific text strings, like city names in a map.
- ***Scripting and animation:*** Vector graphic markup format enables dynamic and interactive graphics far more sophisticated than bitmapped or even Flash™ images.
- ***True XML:*** As an XML language, vector graphic markup language such as VML and SVG offer all the advantages of XML:

- Interoperability
- Internationalization (Unicode support)
- Wide tool support
- Easy manipulation through standard APIs, such as the Document Object Model (DOM) API
- Easy transformation through XML Stylesheet Language Transformation (XSLT).

The following sections will briefly describe VML and SVG, then gather some reasons for choosing VML for this Master's project.

VML

Vector Markup Language (VML) is an XML, text-based markup language. VML has specifications that include features that allow applications to store higher-level application-specific private data within the graphics file. These features promote a higher-level interchange of graphics between applications. It also provides new ways of combining scripting with the Document Object Model (DOM) API to control a Web page's graphical elements. It provides an easy, standard way for a script writer to manipulate the graphic without requiring the use of special software tools.

VML is supported by Internet Explorer version 5.0 or greater. Many software applications, such as Microsoft Word, Excel, and Power Point can automatically convert pictures in vector graphic format by outputting the XML file containing VML.

SVG

Scalable Vector Graphics (SVG) is a new graphics file format and Web development language based on XML. SVG enables Web developers and designers to create dynamically generated, high-quality graphics from real-time data with precise structural and visual control. With this powerful new technology, SVG developers can create a new generation of Web applications based on data-driven, interactive, and personalized graphics. SVG was created by the World Wide Web Consortium (W3C), and currently over twenty organizations, including Sun Microsystems, Adobe, Apple, IBM, and Kodak, have been involved in defining SVG.

Reasons for chosen VML for my Master's project

The main reasons for choosing VML as my target language is that VML is now natively supported by Internet Explorer 5.0 or above. This makes it possible to display MathML by using stylesheet transformation to render it in VML without using any plugins. Even though SVG is an international standard language, it is not yet supported by any of the popular browsers. There are several tools available for displaying MathML through SVG, but in order to be able to render it in popular browsers, the user still needs some kinds of plugins.

It is true that SVG seems to be better, cleaner, and a more complete standard, but considering the purpose of this project that only needs the use of the vector graphic to draw mathematical expression, which VML's capabilities are more than sufficient to accomplish this project.

1.2.2) Deliverable 2: To reproduce by hand in pages 130 and 131 of “A Guide to LaTeX2e” in LaTeX and in MathML.

The purpose of this deliverable was to gain experience with the most common tags of LaTeX and MathML and know what they output in common translating mechanisms as I need to keep that in mind and try to create the comparable or better results in the final stage.

I wrote a LaTeX document and MathML document for the pages 130 and 131 of “A Guide to LaTeX2e” book. Looking at the results, LaTeX documents that are viewed by a DVI viewer gave a better rendering result when compared to the MathML document viewed by the Amaya browser. With the Amaya browser, some of the MathML elements give an unsatisfying rendering such as matrix brackets that have unconnected lines as shown below.

$$\left(\begin{array}{cc|c} x_{11} & x_{12} & \\ x_{21} & x_{22} & \\ & & Y \\ & & Z \end{array} \right)$$

MathML

MathML is the standard XML application used for displaying mathematical notation and content on the Web. The goal of MathML is to enable mathematics to be served, received, and processed on the Web, just as HTML has for text. MathML provides the ability to control the presentation and the meaning of such expressions. It does this by providing two sets of markup tags, one set presents the notation of mathematical data

in markup format (presentation mode), and the other set relays the semantic meaning of mathematical expressions (content mode), enabling complex mathematical and scientific notation to be encoded in an explicit way

LaTeX2e

The history of LaTeX2e begins long ago with the program called TeX. TeX, created by D. E. Knuth, is a computer program for typesetting documents. It takes a suitably prepared computer file and converts it to a form that may be printed on many types of printers, including dot-matrix printers, laser printers and high-resolution typesetting machines.

LaTeX, written by L. B. Lamport, is one of a number of variation of TeX. It is particularly suited to the production of long articles and books, since it has facilities for the automatic numbering of chapters, sections, theorems, equations etc., and also has facilities for cross-referencing. LaTeX2e is a new version of LaTeX and is a standard program for producing mathematical documents.

1.2.3) Deliverable 3: Create a DTD for drawing matrices. Write a XSL transform to render this language in VML.

The purpose of this deliverable was to learn the structure of the markup language by creating my own DTD for drawing matrices. This will help me fully understand the structure of the MathML and LaTeX languages. In this deliverable, I also started to

explore ways to make a transformation from XML to HTML with VML embedded tags using XSL style sheet transformation language.

For this deliverable, a Data Type Definition (DTD) was given for matrices and an example document was written in this language. An XSLT document is then used to translate this particular XML document to VML document for viewing.

The main reason for using the author's own DTD file was to reduce the complexity in the XSLT transformation at this beginning stage of the project. Eventually, the MathML document will be translate to VML using XSLT.

One of the coding difficulties in this deliverable was to allow each row in the matrix to have a different number of column. This problem exists in presentation mode only, since it allows each row in the same matrix to have any number of columns. On the other hand, content mode in MathML has stricter rules that will not allow this type of issue. From my experience with this deliverable, I discovered that implementing a stylesheet transformation from the presentation mode in MathML is more difficult than that of the content mode. Since presentation mode does not have any rules to control the semantic meaning of the mathematical equation, it allows any kind of rendering even though the expression does not make any sense in the real mathematical environment. This makes the transformation harder because, we have to deal with all the possible cases of rendering. It is obvious that if MathML's presentation mode stylesheet transformation can be successfully accomplished, the content mode transformation can be implemented with ease.

Example Result

Below is an example result of a nested matrix in MathML presentation mode:

$$\left[\begin{array}{c} x \quad y \quad 12 \quad y \\ z \left[\begin{array}{c} x \quad y \quad 12 \\ z \quad 1 \quad 8 \\ 90 \quad f \quad g \end{array} \right] \quad 8 \quad 3.4 \quad 3.4 \quad 12 \\ \\ 90 \quad f \quad \left[\begin{array}{c} 1.1 \quad 1.2 \quad 1.3 \quad 1.4 \quad 1.5 \quad 1.6 \\ 2.1 \left[\begin{array}{c} 2.2.1.1 \quad 2.2.1.2 \quad 2.2.1.3 \\ 2.2.2.1 \quad 2.2.2.2 \quad 2.2.2.3 \quad 2.2.2.4 \quad 2.2.2.5 \\ 2.2.3.1 \quad 2.2.3.2 \quad 2.2.3.3 \end{array} \right] \quad 2.3 \quad 2.4 \quad 2.5 \\ 3.1 \quad 3.2 \quad 3.3 \quad 3.4 \quad 3.5 \end{array} \right] \end{array} \right]$$

1.2.4) Deliverable 4 : Get the MathML matrix related, apply, minus, times, divide, and eq tags to translate to VML and to SVG via XSLT.

The purpose of this deliverable was to show that I am ready and have all the necessary knowledge to be able to produce the transformation of some MathML tags to HTML file with embedded VML. This is the beginning stage of the real project I am going to deliver next semester. In this deliverable, I get a chance to fully explore the stylesheet transformation language and the results it produces. I took notes on good and bad coding techniques I tried in this stage as a way to improve the results and the coding style in the final project.

This deliverable gives a stylesheet transformation to VML for some particular tags in MathML language such as: matrix related tags, apply, minus, times, divide, and eq. It differs from the last deliverable in that it translates those tags from the content markup mode, unlike the last deliverable that transforms according to the presentation markup mode.

The use of content markup rather than presentation markup for mathematics is sometimes referred to as semantic tagging. The parse-tree of a valid element structure using MathML content elements corresponds directly to the expression tree of the underlying mathematical expression. However, even in such simple expressions as “X + Y”, some additional information may be required for applications such as computer algebra. Are “X” and “Y” integers, functions, etc.? For example, do we have “X+Y” or just “+X”. The interpretation must determine which operand a given operator should be applied to. This additional information is referred to as semantic mapping. In MathML, this mapping is provided by the semantics, annotation and annotation-xml elements. The semantic elements are the container elements for a MathML expression together with its semantic mappings. Semantic elements expect a variable number of child elements. The first is the element (which may itself be a complex element structure) for which this additional semantic information is being defined. The second and subsequent children, if any, are instances of the elements annotation and/or annotation-xml. Our stylesheet transformation interprets the MathML expression according to some features in the content markup mode explained above, such as the ability to determine which operand should the operator apply to (eg. A-B or -B). Below is an example result of the basic operators’ and matrix’s rendering in MathML content mode:

$$a/10/20$$

$$+ c$$

$$c \times 20 \times 30$$

$$c = 40 = 50$$

$$c - 60 - 70$$

$$\left[\begin{array}{cc} c + 9 & 10 \ 11 \\ \left[\begin{array}{ccc} 1 & 0 & 0 \\ 0 & 1 & 1 \\ 1 & 0 & 1 \end{array} \right] & 12 \ 13 \\ 10 & 11 \ 12 \end{array} \right]$$

2) Requirement

This section discusses the scope of the work in CS298 and the final result of the Master Project. There are many transformation programs that convert from LaTeX to MathML currently available. With all the considerations of the time and the large scope of the LaTeX language, I think it is not necessary to develop a stylesheet transformation to convert LaTeX directly into VML. The LaTeX user can just use one of the LaTeX – MathML conversion program and then apply this project's XSLT transformation to render MathML on the Web.

For MathML users, there are two ways of encoding mathematical data using MathML: Content Markup or Presentation Markup. Content markup is concerned with the semantics of mathematics. Presentation markup is concerned with the rendering of mathematics. Currently, there are also several software applications that help generate MathML easily with powerful Graphic User Interface. Most of these programs generate the MathML in presentation mode, which allows the user to display most of the mathematical expression. However, there is a MathML Content2Presentation Transformation (MathMLc2p), written in XSLT, able to translate content markup expressions into presentation markup expressions automatically.

Within the three month period in CS298 Master Writing Project Course, the goal of this Master's project was to develop a stylesheet transformation from MathML in presentation mode with default rendering and some important attributes into VML. In particular, all the necessary mathematical rendering transformation will be developed; however, some additional attributes that are used to enhance the rendering such as the

spacing attributes or the ability to split an expression if it is too long will be ignored, since this kind of rendering is software/device dependent.

Another intention of this project was for the client side to process the translation.

However, if the translation is done client-side then one is essentially sending the compiler along with the document, which makes it hard to sell this kind of product.

To handle this issue, we will investigate ways to make the stylesheets sent only to the applicable given document requested.

3) Design

The purpose of this project is to demonstrate the feasibility of rendering MathML 2.0 markup with VML, Vector Graphic Markup Language and HTML by using XSLT. It is not intended as a full featured implementation. This project does the style sheet transformation of the MathML file on the fly and produces an HTML file that contains the VML code as output. This allows the user to be able to view the resulting file by using popular browsers such as Microsoft's Internet Explorer.

This translation supports only Presentation elements of MathML and some of their attributes, none of the Content elements are supported. Before we go deep into the design of the translation, let's look at the design of MathML presentation mode first.

The presentation elements are divided into two main groups, token elements group and layout schemata elements group. Token elements represent individual symbols, name, numbers, etc. The content of the token elements can only be characters. All individual symbols in a mathematical expression should be represented by MathML token elements (base elements).

In MathML, expressions are constructed out of smaller expressions, and ultimately out of single symbols, with the parts grouped and positioned using one of a small set of notational structures, which can be thought of as expression constructors. The layout schemata specify the way in which sub-expressions are built into larger expressions. The summary of all the presentation elements is on the next page.

Summary of Presentation Elements

	Element Name	Description
Token Elements		
1.	mi	Identifier
2.	mn	Number
3.	mo	Operator
4.	mtext	Text
5.	mspace	Space
6.	ms	String literal
7.	mglyph	Adding new character glyph to MathML
General Layout Schemata Element		
8.	mrow	Group any number of sub-expressions horizontally
9.	mfrac	Form a fraction from two sub-expressions
10.	msqrt	Form a square root (radical without an index)
11.	mroot	Form a radical with specified index
12.	mstyle	Style change
13.	merror	Enclose a syntax error message from a preprocessor
14.	mpadded	Adjust space around content
15.	mphantom	Make content invisible but preserve its size
16.	mfenced	Surround content with a pair of fences

17.	menclose	Enclose content with a stretching symbol such as a long division sign.
Script and Limit Schemata Element		
18.	msub	Attach a subscript to a base
19.	msup	Attach a superscript to a base
20.	msubsup	Attach a subscript-superscript pair to a base
21.	munder	Attach an underscript to a base
22.	mover	Attach an overscript to a base
23.	munderover	Attach an underscript-overscript pair to a base
24.	mmultiscripts	Attach prescripts and tensor indices to a base
Tables and Matrices Element		
25.	mtable	table or matrix
26.	mlabeledtr	row in a table or matrix with a label or equation number
27.	mtr	row in a table or matrix
28.	mtd	one entry in a table or matrix
29.	maligngroup	alignment markers
30.	malignmark	alignment markers
Actions Element		
31.	maction	bind actions to a sub-expression

In doing this project, there are many situations to be considered. Since our desire is to render MathML without using any kinds of plugins when using XSLT as the only translation method, we have to deal with many kinds of problems due to the weakness of XSLT, compared to more powerful programming language. For instance, the ability to do for-loop, max, min, or even change the value of the variable within the same function are difficult in XSLT. Since, there is no way to keep variables or change the value of a predefined variable, it is impossible to come up with the algorithm that finds the depth of some nested element to render it correctly using only XSLT. Therefore, we require some of the nested tags to specify their length and also using the method of putting everything in the HTML table as well as use the VML coordinate setting within the HTML table for all of the elements translation. With this method, we are able to render the mathematical expression correctly.

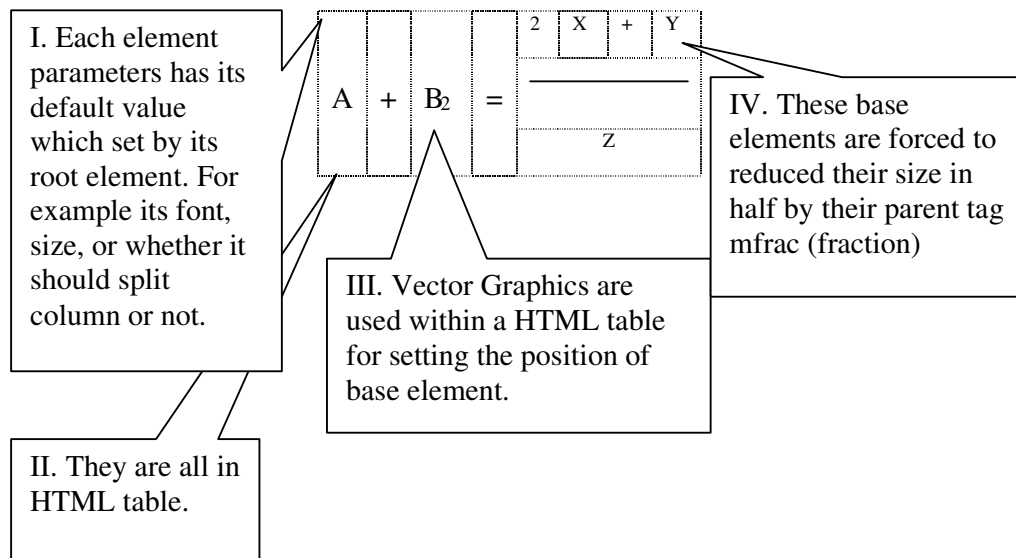
First, let start with the very brief ideas for each part of the design this section and the detailed description and examples will be presented in the following section.

The main design ideas for the translation are:

- I. Hierarchy value passing design: The characteristics or attributes defined by the parent tag are sent to its children through parameters. These important parameters that specify certain characteristics are parsed from the root node to its descendant nodes. This design is forced by the usage of XSLT language.
- II. Table design: The entire tag rendering are contained in one big nested HTML table. This is very useful for adjusting the position of many tags rendering, the likes of fractions, tables, over scripts, under scripts, and etc. This part use the help of HTML language such as the align function of HTML table to adjust the position to be in the center, left, or right.

- III. Vector graphic design: VML graphic or VML language is used for aligning the mathematical expression in the specific position set by vector graphic coordinates and functions. This allows the translation to set the certain sizes, position, font, and aligning for each token element according to its parent tags.
- IV. MathML design: Lastly, the translations are based on the MathML 2.0 rules. The parameters are changed according to the type of MathML tag that it matches. These adjusted parameters will be sent from its parent to its children and ultimately down to the token elements or the base elements such that they will be rendered correctly according to the rule of their parent tag.

The picture below show the brief idea of all 4 main part of the design combined.



I. Hierarchy Value Passing Design.

Since the natural structure of XSLT translation uses a concept of a tree, our design will define a group of configuration parameters used for controlling each element rendering at the root node and pass them down to its descendant as it travel through the MathML input document. Examples of these parameters are: Font, Style, Size,

X_Coordinate, Y_Coordinate, Change_Style, SplitColumn, etc. Each token element has the same default value for these parameters which has been set at the root node such as the size = 25, x-coordinate start at 0 , y-coordinate start at 0, New Courier Font, etc. Then, these parameters' values can be adjusted according to each tag that the translation matches.

For example, we can render MI as follow.

The below is MathML example source document for “mi”.

```
<math>
  <mi>A</mi>
  <msqrt>
    <mi>x</mi>
  </msqrt>
</math>
```

In this translation, once the translation maps the first “mi”, it will match the “mi” template and receive the parameters which in this case will render elements with the predefined default value passed from the root node, such as 25 pixels size, rendered at the start x_coordinate 0 and start y_coordinate 0. On the other hand, when it maps the msqrt tag, it will match msqrt tag which will reduce the parameter size in half and pass it to its child node, which in this case is the “mi” element. This makes mi element render with the size in half of the original default sizes.

II. Table Design:

The fact that XSLT can only pass parameters from the root down to its descendant, but not the other way around, prevents us from keeping track of the total number of some particular group of element as the translation goes down to the descendant node. These numbers are important for rendering some tag such as mfrac (fraction) tag that

need to know whether the number of elements in the numerator or in the denominator is larger, so we can calculate the center position to render the shorter expression correctly.

Let's take a look at how to render mfrac. The rule for rendering mfrac is to take the first child as numerator and the second child as denominator.

The example below should render as: $\frac{x}{y}$

```
<math>
  <mfrac>
    <mi>x</mi>
    <mi>y</mi>
  </mfrac>
</math>
```

This kind of tag would not give any problem in rendering, because the numerator and the denominator have the same length. However, in the case that there are nested mfrac or fraction over fraction, we will have a problem of finding the number with the longest expression, so we could use that number to set the position of the shorter expression to be at the center and to draw the fraction line to cover the length of the longest expression. As we will see in the next example:

The example below should render as:

```
<math>
<mfrac>
  <mrow>
    <mfrac>
      <mrow>
        <mi>x</mi>
        <mo>+</mo>
        <mi>y</mi>
      </mrow>
      <mn>2</mn>
    </mfrac>
  <mo>+</mo>
  <mi>z</mi>
</math>
```

```

</mrow>
<mfrac>
  <mrow>
    <mi>A</mi>
    <mo>+</mo>
    <mn>3</mn>
  </mrow>
  <mn>6</mn>
</mfrac>
</math>

```

In this case, we would have to go through the first child of the first mfrac to find the length of its numerator. This has to go through the nested mfrac and check whether the numerator or the denominator is longer than the length of the parent mfrac's numerator. Unfortunately, I found out that XSLT supports the ability of changing a variable only very poorly. Although XSLT does provide a way to change the value of a variable through a recursive function, it does not allow for any recursive node passing algorithm, which prevents our design to store or update the count value as it goes down to the descendant level.

This is the main reason for putting every element rendering in the HTML table, so that we can use certain HTML features, such as “align = ‘center’” to force the expression in the fraction to render in the center without knowing the length of either the numerator or the denominator.

III. Vector Graphic Design:

Even though all elements are contained in a table structure, the content inside every table item is set by the VML (Vector graphic) group coordinate system. This method allow the translation to set the size and the fixed “x” and “y” coordinate position of

each element, or to draw mathematical symbol at the desired position. This is very useful for the nested tag that we may have to reduce the size of the element or calculate the new position of the “x” and “y” coordinates recursively as presented in the next example.

Below is the example of how to render msub

```
<math>
  <msub>
    <mi>x</mi>
    <mn>2</mn>
  </msub>
</math>
```

In this case, once the translation matches msub template, it will check whether it needs to split the column in the table or not by checking to see if ‘splitColumn’ parameter equals to true (in this case, it must be true, because its parent is the root element <math> which set the default value of split column to be true). After adding the necessary tag to the add column in the table, it will set the VML group coordinate just for render the subscript with base x and subscript 2. It will map the first child element and render it as a base element with the original size and original x and y coordinate (0,0). Then, it will find the second child and render it as a subscript with the size reduced in half, the new x coordinate equals the original plus the size of the base element, and the new y coordinate equals the original plus the size divided by 2 to get the new location of subscript which must be lower than the base element. Once it calculates the necessary new position of the base element and the subscript element, these adjusted parameters will be passed to the token element, which in this case is “mi” element and “mn” element, to render itself with these new parameters.

IV. MathML Design:

Our design structure is based on a tree structure as described in the section hierarchy value passing design section; the main parameters that are used for rendering each element will be changed according to the parent tags, which are strictly translated according to the MathML 2.0 rules. As an example in the msub translation, the parameters are passed from the root $\langle\text{math}\rangle$ element through the parent level element, which is $\langle\text{msub}\rangle$, This in turn defines new parameters for rendering its children $\langle\text{mi}\rangle$ and $\langle\text{mn}\rangle$ then passes these parameters down to its children. At the token element or base element level is where the rendering actually takes place by using all of the necessary parameters that are passed from the parent element.

4) Implementation

This section explores the description of each element in MathML presentation markup and explains how this project accomplishes the translation for each element in detail.

The math `<math>` element is the root element for every MathML document. XSLT translation for this project provides the “math” template for mapping the root element of a MathML document. At this math template, necessary headers for rendering HTML and VML code in the result document will be added at this point. All the VML predefined shapes for rendering all the special mathematical symbols and the parameters that are used for rendering each element are also defined here.

The pseudo-code for transforming the root element `<math>` template:

```
<xsl:template match="math">
  Add header for rendering VML.
  Define predefined shapes.
  Add header for rendering HTML.
  <!-- Defined parameters with default value -->
  <xsl:apply-templates select="*">
    <xsl:with-param name="splitColumn" select="1" />
    <xsl:with-param name="set_coordinate" select="0" />
    <xsl:with-param name="x_coordinate" select="0" />
    <xsl:with-param name="y_coordinate" select="0" />
    <xsl:with-param name="y_msub_coordinate"select="0"
    <xsl:with-param name="font" select="'courier new'" />
    <xsl:with-param name="size" select="25" />
    <xsl:with-param name="color" select="'black'" />
  </xsl:apply-templates>
</xsl:template>
```

The presentation elements are divided into five classes, token elements, general layout schemata elements, script and limit schemata element, table and matrices, and action

element. The translations of every element in both classes are described in the following sections.

Token elements

Token elements include `mi`, `mn`, `mo`, `mtext`, `mSPACE`, `ms`, and `mglyph`. Elements in this category are called base elements in which they can not have any sub element nested inside them. The value of token element should be rendered according to the type of the element with regards to the font, size, style, etc. For example the default rendering for `mi` would be “italic” style with “Courier New” font.

For the regular rendering of these token elements to be rendered in one column in the table for each element, and inside each column, we also specify the VML group coordinate to render each token element with the length according to the length of each element value.

The rendering of these token elements can be altered depending on the parent element of the particular token. This could be done to specify whether it should split the column in the table before rendering or it should use the predefined coordinate that parsed from the parent element or not.

The brief description of each token element, its specific rendering details, and examples are provided as follow:

mi

An mi element represents a symbolic name or arbitrary text that should be rendered as an identifier. Identifiers can include variables, function names, and symbolic constants. The default rendering for mi is in the “italic” style and “Courier New” font.

Example: A x

```
<mi> A </mi>
<mi> x </mi>
```

As we have discussed on the design for this project in the design section, each element rendering is based on the configuration parameters passed down from its parent. Every transformation templates for each element has a similar format as shown below in the high level design for mi element.

Below some pseudo-code for transforming an mi element:

```
<xsl:template match="mi">
  <!-- Get configuration parameters from the parent node.-->
  <xsl:param name="splitColumn"/>
  <xsl:param name="set_coordinate"/>
  <xsl:param name="x_coordinate"/>
  <xsl:param name="y_coordinate"/>
  <xsl:param name="font"/>
  <xsl:param name="size"/>
  <xsl:param name="color"/>

  If $$splitColumn = true Then
  {
    Splilt Column by adding HTML "td" tag.
  }
  If $$Set_coordinate = true Then
  {
    Set the coordinate for rendering VML within the same group.
  }
  Render this element with parameters passed from its parent.
</xsl:template>
```

mn

An mn element represents a numeric literal such as a sequence of digits, a decimal point, unsigned or signed integer or real number. The default rendering for mn is in the “normal” style and “Courier New” font.

Examples: 2 0.123 1,000,000 2.1e10 0xFFWF

```
<mn> 2 </mn>
<mn> 0.123 </mn>
<mn> 1,000,000 </mn>
<mn> 2.1e10 </mn>
<mn> 0xFFEF </mn>
```

The high level design for transforming mn element is similar to mi template except for the style for mn should be normal not italic.

mo

An mo element represents an operator, symbols, and notations that are not mathematical operators in the ordinary sense. It includes fence characters such as braces, parentheses, and ‘absolute value’ bars, symbols such as right arrows, left arrows, separators such as comma and semicolon, and mathematical accents such as a bar or tilde over a symbol. Invisible character include ⁢ or ⁢ (i.e. xy) , ⁡ or ⁡ (i.e. f(x) sin x), and ⁣ or ⁣ (i.e. m12) will be detected mo template and skip its rendering.

The default rendering for mo is in the “normal” style and “Courier New” font. Special mathematical symbols that are added in the MathML file should also define an internal entity declaration at the top of the MathML file. Later, at the translation time,

the mo template must have a function to check whether that symbol has already been defined in the vector graphic predefined shape or not by consulting a table or library.

Example: $3 < (X+5)$

```
<mn>3</mn>
<mo>&lt;</mo>
<mrow>
  <mo>(</mo>
  <mi>X</mi>
  <mo>+</mo>
  <mn>5</mn>
  <mo>)</mo>
</mrow>
```

The pseudo-code for transforming mo element:

```
<xsl:template match="mo">
  Get configuration parameters from the parent node.

  If $SplitColumn = true Then
  {
    Spilt Column by adding HTML "td" tag.
  }
  If $ Set_coordinate = true Then
  {
    Set the coordinate for rendering VML within the same group.
  }
  Use the operator name to create an id to call the VML predefined shape for
  that operator by consulting the predefined shape table or library.
  Render this element with parameters passed from its parent.
</xsl:template>
```

mtext

An `mtext` element is used to represent arbitrary text that should be rendered as itself.

In general, the `mtext` element is intended to denote commentary text. The default rendering for `mtext` is in the “normal” style and “Courier New” font.

Example: This will render as “maps to”.

```
<mtext>maps to</mtext>
```

High level design for transforming mn element is similar to mtext template.

mspace

An mspace empty element represents a blank space of any desired size, as set by its attributes. The mspace element is generally used with one or more attribute values explicitly specified.

There are four main attributes for mspace element which are width, height, depth, and linebreak. This project will only provide the rendering of width attribute. The h-unit and v-unit represent units of horizontal or vertical length for width, height, and depth. In MathML, as in XML, 'whitespace' means simple spaces, tabs, newlines, or carriage returns, i.e., characters with hexadecimal Unicode codes U+0020, U+0009, U+000A, or U+000D, respectively.

I only implemented the width attribute as an example here. The brief design for implementing a width attribute for mspace is to get the horizontal unit value from the width attribute and render them as whitespace symbol with the width of the specified width from its attribute.

Example: This will render as 2 blank spaces.

```
<mspace width="2em"/>
```

The pseudo-code for transforming mspace:

Template mspace:

```
{
    Get configuration parameters from the parent node.

    $width = width from its width attribute.
    If (splitColumn = true ) Then
    {
        add HTML <td> tag around blank.
    }
}
```

```

    If (set_coordinate = true) Then
    {
        Add necessary VML group tag to set its coordinate.
    }
    Render black symbol with the width of $width.
}

```

ms

The ms element is used to represent 'string literals' in expressions meant to be interpreted by computer algebra systems or other systems containing 'programming languages'. For example, `<ms><<</ms>` represents a string literal containing 5 characters, the first one of which is &.

The high level design for rendering ms element is similar to mi template, except for the part that the translation must render them as disable output escaping element. This way the element will be render as a string as it appear in the markup element and does not render it using the reference entity.

mglyph

The mglyph element is the means by which users can directly access glyphs for characters that are not defined by Unicode, or not known to the renderer.

Example

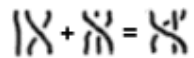
The following example illustrates how a researcher might use the mglyph construct with an experimental font to work with braid group notation.

```

<mrow>
  <mi><mglyph fontfamily="my-braid-font" index="2" alt="23braid"/></mi>
  <mo>+</mo>
  <mi><mglyph fontfamily="my-braid-font" index="5" alt="132braid"/></mi>
  <mo>=</mo>
  <mi><mglyph fontfamily="my-braid-font" index="3" alt="13braid"/></mi>
</mrow>

```

This might render as:



The idea for translating `mglyph` is to create a predefined table of glyph with fontfamily and index in XSLT and consult that table every time it matches `mglyph` element. The `alt` attribute provides an alternate name for the glyph. In the case that the specified font cannot be found, XSLT can use this name in a warning message or some unknown glyph notation. The font family and index uniquely identify the `mglyph`. The `alt` attribute should not be part of the identity test.

The pseudo-code for `mglyph` element:

Template `mglyph`:

```
{
  Get configuration parameters from the parent node.
  $glyph = Get the glyph name from its attributes.

  If ($glyph is in the glyph library) then
  {
    Render the glyph using VML predefined shape $glyph.
  }
  Else DisplayErrorMessage().
}
```

General Layout Schemata

A general layout schema includes `mrow`, `mfrac`, `msqrt`, `mroot`, `mstyle`, `merror`, `mpadded`, `mphantom`, `mfenced`, and `menclosed`.

`mrow`

An `mrow` element is used for grouping any number of sub-expressions together, usually consisting of one or more `mo` elements acting as operators on one or more other expressions that are their operands.

When an mrow is needed to define arguments to another MathML element such as the case of mfrac element, that models a fraction which expects two arguments: a numerator and a denominator, the translation will use mrow as a path to send necessary set of configuration parameters down to its children.

Example, a + b divided by 2:

```
<mfrac>
  <mrow>
    <mi>a</mi><mo>+</mo><mi>b</mi>
  </mrow>
  <mn>2</mn>
</mfrac>
```

The design for translating mrow element is just simply makes the mapping of mrow be a path which passes necessary set of configuration parameters from the parent node down to the children level of mrow element.

mfrac

The mfrac element is used for fractions. The syntax for mfrac is `<mfrac> numerator denominator </mfrac>`

There are two types of mfrac translation the first one is for the mfrac that contain no nested sup expression. The second type is for translating mfrac which has nested sub expression.

The brief design for the first type is described as follow:

1. Match the first child that represent numerator display it on the top part
2. Draw the fraction line in between numerator and denominator with the length of numerator or denominator size.

3. Match the second child that representing denominator and display it at the bottom with the size reduced to half.

The design for the mfrac that has nested sub expressions is described below:

1. This one requires the tag to specify the length of its numerator and denominator.
2. Once we know whether the numerator or denominator is longer, we can precisely draw the fraction line with the length long enough to cover fraction.
3. We put the entire fraction in nested HTML table tag, so that numerator, fraction line, and denominator expression will align in the center.

TEMPLATE “MFRAC”: Render fraction that has only one level of numerator and denominator.

The pseudo-code:

```
Template mfrac: // for one non-nested mfrac.  
{  
    Get configuration parameters from the parent node.  
    Match the first child for rendering numerator.  
    Draw fraction line.  
    Match the second child for rendering denominator.  
}
```

TEMPLATE “MFRAC[NESTED ELEMENT]” Render fractions that has nested sub expression in the numerator or denominator or both.

High Level Design:

```

Template mfrac: // for nested mfrac.
{
    Get configuration parameters from the parent node.

    Add HTML <table align= "center" > tag.
    { Add HTML <tr> tag. // add first row.
        {
            // Render numerator.
            Add HTML <table align = "center" > tag. // put in the table so it will
            automatically adjust itself to be in the center.
            { Add HTML <tr> tag. //add row.
                { Match the first child for rendering numerator with size reduced
                to half and set "split column = true".
            }
        }
    }

    Add HTML <tr> tag. // add second row.
    { If ( $numerator > $denominator ) Then
        {
            Add HTML <td> <div align = "center"> tag. // add column.
            { Draw line with the length = $size * $numerator.
            }
        }
        Else
        {
            Add HTML <td> <div align = "center"> tag. // add column.
            { Draw line with the length = $size * $numerator.
            }
        }
    }

    Add HTML <tr> tag. // add second row.
    { // Render denominator.
        Add HTML <table align = "center" > tag.
        { Add HTML <tr> tag. //add row.
            { Match the second child for rendering denominator with size
            reduced to half and set "split column = true".}
        }
    }
}
}

```

msqrt

These elements construct radicals. The msqrt element is used for square roots The syntax for msqrt elements is:

<msqrt> base </msqrt>

Example:

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

```
<msqrt>
  <mrow>
    <msup>
      <mi>b</mi>
      <mn>2</mn>
    </msup>
    <mo>-</mo>
  <mrow>
    <mn>4</mn>
    <mi>a</mi>
    <mi>c</mi>
  </mrow>
</mrow>
</msqrt>
```

According to the limitation of XSLT language, I divided the translation into two types.

The first type is the perfect rendering, which only applies to the token element rendering in the msqrt tag.

The second type is for the msqrt tag that contains non-token tag such as msqrt itself, mfrac, mroot, etc. For this second type, I require the msqrt tag to have the width and height attribute. This way allows the translation to know exactly the width and the height of the nested expression inside the square root symbol to be able to draw the square root symbol correctly. If the tag does not provide the width and the height of the nested expression, there is no way to find out using only XSLT, because of the limitation of the language. XSLT only provide a way to hold the value in a variable, but it can only be defined once, it does not provide a way to change the value of the same variable as it goes down to the next child or descendant level of the node. I have tried many techniques to try to get the width and the height of the node using only

XSLT, and I found out that there is no way to do so. However, this may be accomplished by the help of DOM or Javascript. For this project, I will not try to use any other technologies besides XSLT, because that will ruin the purpose of this project.

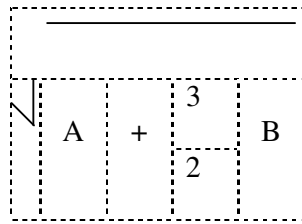
The design of the perfect rendering is:

1. Find the width of the expression inside the square root by counting the total of all token elements. There is no need to find the height of expression, since all of the expressions are token element that means there is no nested element, so they all have the same size which is the height of one character.
2. Assign the start x-coordinate position of each sub element according to its position. They are separated by the same predefined gap.
3. Draw the square root symbol according to the width and the height collected by the first step.
4. Call apply-templates according to its position with the match start x-coordinate position defined by step 2 and the right y-coordinate and the original size divide by 2.

The design of the second type of nested sub expression inside msqrt tag is described below:

1. Get the width and the height of the nested sub-expression from msqrt's width and height attributes.

- Split the columns and rows as needed to draw the square root symbol as in this picture. (Note that this will give an imperfect shape of square root symbol.)



- Every time it calls apply-template according to the type of the element tag, it needs to split the columns. This way every element will be rendered in the appropriate alignment by the property of HTML table tag. This will avoid making a coordinate setting for each tag which will be impossible to do using only XSLT.

TEMPLATE " MSQRT: Render square root without nested sub expression. Calculate the width of expression by counting all token elements in the tag. Draw the square root symbol according to the width. Call matching element template with predefined coordinate position for each child element.

The pseudo-code:

Template msqrt: // for msqrt that has only one level of children

```
{
  Get configuration parameters from the parent node.
  $width =count( mrow[(mi + mo + mn ) ])
  Assign the position to each child element.

  Draw square root symbol with the width according to the length of sub
  expression.

  Call matching child template with appropriate coordinates, and size/2.
}
```

TEMPLATE "MSQRT[//not token elements]": Render square root with nested sub element. Get the width and height from its attribute. Draw the square root symbol

according to the width and height. Split column as necessary every time it call apply-templates for its nested elements.

```

Template msqrt(decendant != token element ) //for msqrt that has nested children
{
    Get configuration parameters from the parent node.
    $width = Get the width from its width attribute.

    Add HTML <table> tag
    {
        Add HTML <tr> tag // add row.
        {
            Add HTML <td colspan = $width> tag //add column with
            columnspans equal to the length of the entire sub expression.
            {
                Draw top part of square root symbol.(  $\sqrt{\quad}$  )
            }
        }

        Add HTML <tr> tag // add new row.
        {
            Add HTML <td> tag // add first column to second row.
            {
                Draw front part of the square root symbol. (  $\sqrt{\quad}$  )
            }

            Call matching child template with appropriate coordinates,
            size/2, and parameter 'splitcolumn = true'.
        }
    }
}

```

mroot

The mroot element is used to draw radicals with indices, e.g. a cube root. This syntax for mroot element is:

```
<mroot> base index </mroot>
```

The mroot element requires exactly 2 arguments.

The translation design for MROOT is similar to MSQRT except that we also have to draw the root index for this translation.

TEMPLATE " MROOT: Render square root without nested sub expression. Calculate the width of expression by counting all token elements in the tag. Draw the square root symbol according to the width and its index. Call matching element template with predefined coordinate position for each child element.

The pseudo-code:

```

Template mroot: // for mroot that has only one level of children
{
    Get configuration parameters from the parent node.
    $width =count( mrow[(mi + mo + mn ) ] - 1)
    Assign the position to each child element.

    Draw root index by call matching child at position = 2  with
    size = size /3 and appropriate x and y coordinates.

    Draw square root symbol with the width according to the length of sub
    expression.

    Call matching child template with appropriate coordinates with size/2
}

```

TEMPLATE "MROOT[//not token elements]": Render square root with nested sub element. Get the width and height from its attribute. Draw the square root symbol according to the width and height. Split column as necessary every time it call apply-templates for its nested elements.

```

Template mroot(decendant != token element ) //for mroot that has nested children
{
    Get configuration parameters from the parent node.
    $width = Get the width from its width attribute.

    Add HTML <table> tag
    {
        Add HTML <tr> tag // add row.
        {
            Add HTML <td colspan= $width> tag //add column with
            columnspans equal to the length of the entire sub expression.
            {

```



```

        Draw top part of square root symbol.(  $\sqrt{\quad}$  )
    }
}

Add HTML <tr> tag // add new row.
{
    Draw root index by call matching child at position = 2 with
    size = size / 3 and appropriate x and y coordinates.

    Add HTML <td> tag // add first column to second row.
    {
        Drow front part of the square root symbol. (  $\sqrt{\quad}$  )
    }

    Call matching child template with appropriate coordinates,
    size/2, and parameter 'splitcolumn = true'.
}
}

```

mstyle

The mstyle element is used to make style changes that affect the rendering of its contents. mstyle can be given any attribute accepted by any MathML presentation element provided that the attribute value is inherited, computed or has a default value; presentation element attributes whose values are required are not accepted by the mstyle element.

Examples: The example of limiting the stretchiness of a parenthesis shown in the section on <mo>.

```

<mrow>
  <mo maxsize="1"> ( </mo>
  <mfrac> <mi> a </mi> <mi> b </mi> </mfrac>
  <mo maxsize="1"> ) </mo>
</mrow>

```

can be rewritten using mstyle as:

```

<mstyle maxsize="1">
  <mrow>
    <mo> ( </mo>
    <mfrac> <mi> a </mi> <mi> b </mi> </mfrac>
    <mo> ) </mo>
  </mrow>
</mstyle>

```

The design for translating mstyle element is to check the type of mstyle's attribute and set the configuration parameters accordingly, and then pass these parameters down to the child level.

merror

The merror element displays its contents as an 'error message'. This might be done, for example, by displaying the contents in red, flashing the contents, or changing the background color. The contents can be any expression or expression sequence.

Example: All these tag within <merror> tag will be render in red.

```
<merror>
  <mtext> Unrecognized element: mfraction;
    arguments were: </mtext>
  <mrow> <mn> 1 </mn> <mo> + </mo> <msqrt> <mn> 5 </mn> </msqrt>
</mrow>
  <mtext> and </mtext>
  <mn> 2 </mn>
</merror>
```

The translation for merror is just simply changing the color parameter to red, and then passing it down to its children element. This will render the elements inside merror tag in red.

mpadded

An mpadded element renders the same as its content, but with its overall size and other dimensions modified according to its attributes. The name of the element reflects the use of mpadded to effectively add 'padding', or extra space, around its content. The mpadded element does not rescale its content; its only effect is to modify the apparent size and position of the 'bounding box' around its content.

Examples: They are all render as "...".

```
<mpadded width="+0em"> ... </mpadded>
<mpadded width="+0%"> ... </mpadded>
<mpadded width="1 width"> ... </mpadded>
<mpadded> ... </mpadded>
```

The idea for translating mpadded is simply add the blank spaces around its content with the specified size according to its attribute.

mphantom

The mphantom element renders invisibly, but with the same size and other dimensions, including baseline position that its contents would have if they were rendered normally. mphantom can be used to align parts of an expression by invisibly duplicating sub-expressions.

The mphantom element accepts any number of arguments; if this number is not 1, its contents are treated as a single 'inferred mrow' formed from all its arguments

Example: In this example, mphantom is used to ensure alignment of corresponding parts of the numerator and denominator of a fraction:

```
<mfrac>
  <mrow>
    <mi> x </mi>
    <mo> + </mo>
    <mi> y </mi>
    <mo> + </mo>
    <mi> z </mi>
  </mrow>
  <mrow>
    <mi> x </mi>
    <mphantom>
      <mo> + </mo>
      <mi> y </mi>
    </mphantom>
    <mo> + </mo>
    <mi> z </mi>
  </mrow>
</mfrac>
```

This would render as something like

$$\frac{x + y + z}{x + z}$$

rather than as

$$\frac{x + y + z}{x + z}$$

We just use a very simple method for translating mphantom by just changing the color parameter to the background color and send it down to its children element. This way makes the element invisible but also preserves the actual size of the element.

mfenced

The mfenced element provides a convenient form in which to express common constructs involving fences (i.e. braces, brackets, and parentheses), possibly including separators (such as comma) between the arguments.

Example 1: renders as (x)

```
<mfenced> <mi>x</mi> </mfenced>
```

Example 2: renders as (x,y)

```
<mfenced> <mi>x</mi> <mi>y</mi> </mfenced>
```

The pseudo-code for mfenced:

```
Template mfenced
{
    Get configuration parameters from the parent node.

    Add open fence according to its attribute( default is '(' ).
    Match the first child element with configuration parameters.
    While More children( )
    {
        Add comma ','.
        Match each child with configuration parameters.
    }
    Add close fence according to its attribute( default is ')' ).
}
```

menclose

The `menclose` element renders its content inside the enclosing notation specified by its `notation` attribute. The main notation attribute for `menclose` are `longdiv`, `actuarial`, and `radical`. The default value is `longdiv`. For the `longdiv` notation attribute, the contents are drawn enclosed by a long division symbol. For the `actuarial` notation, the contents are drawn enclosed by an actuarial symbol. The case of `notation=radical` is equivalent to the `msqrt` schema. Example of using `menclosed` for long division problem: it will render as:

$$\begin{array}{r} 10 \\ 131 \overline{)1413} \\ \underline{131} \\ 103 \end{array}$$

```
<table columnspacing='0' rowspacing='0'>
<mtr>
  <td></td>
  <td columnalign='right'><mn>10</mn></td>
</mtr>
<mtr>
  <td columnalign='right'><mn>131</mn></td>
  <td columnalign='right'>
    <menclose notation='longdiv'><mn>1413</mn></menclose>
  </td>
</mtr>
<mtr>
  <td></td>
  <td columnalign='right'>
    <mrow>
      <munder>
        <mn>131</mn>
        <mo> &UnderBar; </mo>
      </munder>
      <mphantom><mn>3</mn></mphantom>
    </mrow>
  </td>
</mtr>
<mtr> <td></td> <td columnalign='right'><mn>103</mn></td>
</mtr>
</table>
```

Example of using menclose for actuarial notation: it will render as:

$$\frac{a}{n | i}$$

```
<msub>
  <mi>a</mi>
  <mrow>
    <menclose notation='actuarial'>
      <mi>n</mi>
    </menclose>
    <mo>&it;</mo>
    <mi>i</mi>
  </mrow>
</msub>
```

The pseudo-code for menceded:

Template menceded

```
{
  Get configuration parameters from the parent node.

  Child_Width = Find the length of its children element( )
  If Attribute notation = 'longdiv' Them
  {
    Draw Longdiv symbol with the width of Child_Width.
  }
  If Attribute notation = 'actuarial' Them
  {
    Draw Actuarial symbol with the width of Child_Width.
  }
  Match each child with configuration parameters.
}
```

Script and Limit Schemata

Script and limit schemata include msub, msup, msubsup, munder, mover, munderover, mmultiscripts

msub

The syntax for the msub element is:

```
<msub> base subscript </msub>
```

Example:

```

<math>
  <msub>
    <mi>a</mi>
    <mi>1</mi>
  </msub>
  <mo>+</mo>
  <msub>
    <mi>a</mi>
    <mi>2</mi>
  </msub>
</math>

```

Since msub element has 2 children: the first child is the base element, the second child is the subscript element, the translation must find the formula for the x and y coordinate for both base element and the subscript element, and then map each child with the appropriate configuration parameters. For example, the base will render with the default size, x and y coordinates, while the subscript render with the size reduced in half, x coordinate plus the length of the base element, and y coordinate plus the original size divided 2. The pseudo-code for msub is provided below:

```

<xsl:template match="msub">
  Get configuration parameters from the parent node.
  Set VML group coordinate to render both base element and its subscript
  within the same coordinate group.
  <!-- Base element -->
  If Child_position = 1 Then
  {
    <xsl:apply-templates select="child::*[position() = 1]">
      <xsl:with-param splitColumn  ="false" />
      <xsl:with-param set_coordinate ="true" />
      <xsl:with-param x_coordinate  ="$x_coordinate" />
      <xsl:with-param y_coordinate  ="$y_coordinate"/>
      <xsl:with-param size          ="$size" />
    </xsl:apply-templates>
  }

  <!-- Subscript element -->
  If Child_position = 2 Then
  {
    <xsl:apply-templates select="child::*[position() = 2]">
      <xsl:with-param splitColumn  ="false" />
      <xsl:with-param set_coordinate ="true" />
      <xsl:with-param x_coordinate  ="$x_coordinate + $size" />

```

```

        <xsl:with-param y_coordinate = "$y_coordinate + ($size/2)"/>
        <xsl:with-param size = "$size/2" />
    </xsl:apply-templates>
}
</xsl:template>

```

msup

The syntax for the msup element is:

```
<msup> base superscript </msup>
```

Example:

```

<math>
  <msup>
    <mi>a</mi>
    <mi>1</mi>
  </msup>
  <mo>+</mo>
  <msup>
    <mi>a</mi>
    <mi>2</mi>
  </msup>
</math>

```

The design for msup element is similar to the design of msub element except for the formula for the y coordinate of the superscript must be subtracted by the original size divided by 2 in order to render the superscript in the position higher than the base element.

mssubsup

The mssubsup element is used to attach both a subscript and superscript to a base expression.

The syntax for the mssubsup element is:

```
<mssubsup> base subscript superscript </mssubsup>
```

Example: x_1^2 .


```

<math>
  <msubsup>
    <mi>x</mi>
    <mn>1</mn>
    <mn>2</mn>
  </msubsup>
</math>

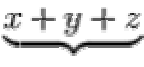
```

The design of this element translation is just simply the combination of the msub and msup with the first child renders as the base element, second child renders as subscript element, and the third child renders as superscript element.

munder

The syntax for the munder element is:

```
<munder> base underscript </munder>
```

Example: 

```

<munder>
  <mrow>
    <mi> x </mi>
    <mo> + </mo>
    <mi> y </mi>
    <mo> + </mo>
    <mi> z </mi>
  </mrow>
  <mo> &UnderBrace; </mo>
</munder>

```

The basic design for munder is to render the base element with the default size and x, y coordinates, while render the under script with the original size divided by 2 and with the x, y coordinate position lower than the base element position. However, there are several cases to consider and handle differently as described below:

- 1) Both base and under script elements are a single token element. In this case, we simply render them as described in the basic design.
- 2) The base is expression and the under script is symbol as in the example above.

In this case, we must find the total length of the base expression and use to be the width of the under script symbol.

3) Both base and under script elements are expression. In this case, we must render them in rows of HTML table using center alignment setting, which will automatically adjust the alignment of the both the base and under script element to be in the center.

mover

The syntax for the mover element is:

```
<mover> base overscript </mover>
```

Example: 


```
<mover accent="true">  
  <mi> x </mi>  
  <mo> &Hat; </mo>  
</mover>
```

The design for mover is very similar to munder's translation design except for the rendering of the over script must be above the base element instead of beneath the base element.

munderover

The syntax for the munderover element is:

```
<munderover> base underscript overscript </munderover>
```

Example: 

```
<munderover>  
  <mo> &int; </mo>  
  <mn> 0 </mn>  
  <mi> &infin; </mi>  
</munderover>
```

The design for munderover is simply the combination of munder and mover translation which the first child is the base element, the second child is the under script element and the third child is the overscript element.

mmultiscripts

The syntax for the mmultiscripts element is:

```
<mmultiscripts>
  base
  (subscript superscript)*
  [ <mprescripts/> (presubscript presuperscript)* ]
</mmultiscripts>
```

Example: R_{ik}^j (where k and l are different indices)

```
<mmultiscripts>
<mi> R </mi>
<mi> i </mi>
<none/>
<none/>
<mi> j </mi>
<mi> k </mi>
<none/>
<mi> l </mi>
<none/>
</mmultiscripts>
```

The design for mmultiscripts is to render the first child as the base element and the even position child as the subscript and the odd position child (except for the first child) as the super script.

The pseudo-code for mmultiscripts:

```
<xsl:template match="msub">
  Get configuration parameters from the parent node.
  Set VML group coordinate to render both base element and its subscript
  within the same coordinate group.
  <!-- Base element -->
  If Child_position = 1 Then
  {
    <xsl:apply-templates select="child::*[position() = 1]">
      <xsl:with-param splitColumn  ="false" />
```

```

        <xsl:with-param set_coordinate ="true" />
        <xsl:with-param x_coordinate  ="$x_coordinate" />
        <xsl:with-param y_coordinate  ="$y_coordinate"/>
        <xsl:with-param size          ="$size" />
    </xsl:apply-templates>
}

<!-- Subscript element -->
If Even_child_position Then
{
    <xsl:apply-templates select="child::*[mod(position()/2) = 0]">
        <xsl:with-param splitColumn  ="false" />
        <xsl:with-param set_coordinate ="true" />
        <xsl:with-param x_coordinate  ="$x_coordinate +
                                [($size/2)*floor(position( )/2)]
        <xsl:with-param y_coordinate ="$y_coordinate + ($size/2)"/>
        <xsl:with-param size          ="$size/2" />
    </xsl:apply-templates>
}

<!-- Superscript element -->
If Odd_child_position Then
{
    <xsl:apply-templates select="child::*[mod(position()/2) = 0]">
        <xsl:with-param splitColumn  ="false" />
        <xsl:with-param set_coordinate ="true" />
        <xsl:with-param x_coordinate  ="$x_coordinate +
                                [($size/2)*floor(position( )/2)]
        <xsl:with-param y_coordinate ="$y_coordinate - ($size/2)"/>
        <xsl:with-param size          ="$size/2" />
    </xsl:apply-templates>
}

</xsl:template>

```

Tables and Matrices

Tables and Matrices include `mtable`, `mlabeledtr`, `mtr`, `mtd`, `maligngroup`, `malignmark`.

mtable

A matrix or table is specified using the `mtable` element. Inside of the `mtable` element, only `mtr` or `mlabeledtr` elements may appear. The column of the table is specified by the `mtr` tag. However, the label in an `mlabeledtr` element is not considered a column in the table.

mtr

An `mtr` element represents one row in a table or matrix. An `mtr` element is only allowed as a direct sub-expression of an `mtable` element, and specifies that its contents should form one row of the table. Each argument of `mtr` is placed in a different column of the table, starting at the leftmost column.

mtd

An `mtd` element represents one entry, or cell, in a table or matrix. An `mtd` element is only allowed as a direct sub-expression of an `mtr` or an `mlabeledtr` element.

Example:

$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

```

<mrow>
  <mo> ( </mo>
  <mtable>
    <mtr>
      <mtd> <mn>1</mn> </mtd>
      <mtd> <mn>0</mn> </mtd>
      <mtd> <mn>0</mn> </mtd>
    </mtr>
    <mtr>
      <mtd> <mn>0</mn> </mtd>
      <mtd> <mn>1</mn> </mtd>
      <mtd> <mn>0</mn> </mtd>
    </mtr>
    <mtr>
      <mtd> <mn>0</mn> </mtd>
      <mtd> <mn>0</mn> </mtd>
      <mtd> <mn>1</mn> </mtd>
    </mtr>
  </mtable>
  <mo> ) </mo>
</mrow>

```

The basic idea for translating mtable, mtr and mtd is to use the methods of HTML table with adjusted rowspans for rendering the left and right bracket that must wrap around all the element in the matrix. This seems to be a simple process but it actually very complicated using XSLT, because it is very hard to find the total number of the elements in each column or row nested inside the mtable tag.

The pseudo-code for mtable:

```

<xsl:template match="mtable">
  Get configuration parameters from the parent node.
  Add the HTML table tag with alignment at center
  If the first child of the mtable specifies the left bracket Then
  {
    Add HTML <tr> tag for adding row.
    {
      Add HTML <td> tag with rowspan = total number of mtr in mtable.
      {
        Render the left bracket with the content of the first child of mtable.
      }
    }
  }
  Matches the child element with configuration parameters.
  If the last child of the mtable specifies the right bracket Then

```

```

    {
      Add HTML <tr> tag for adding row.
      {
        Add HTML <td> tag with rowspan = total number of mtr in mtable.
        {
          Render the left bracket with the content of the first child of mtable.
        }
      }
    }
  }
</xsl:template>

<xsl:template match="mtr">
  Get configuration parameters from the parent node.
  Add HTML <tr> tag for adding row.
  {
    Matches the child element with configuration parameters.
  }
</xsl:template>

<xsl:template match="mtd">
  Get configuration parameters from the parent node.
  Add HTML <td> tag for adding row.
  {
    Matches the child element with configuration parameters.
  }
</xsl:template>

```

mlabeledtr

The `mlabeledtr` element represents a labeled row of a table and can be used for numbered equations. All children of `mlabeledtr` must be `mtd`, where the first child of `mlabeledtr` is the label which located either on the far right or far left of the equation (the default is far right of the equation), and the rest of the child represents the contents of the row.

Example:

$$A = b + c \qquad (2.1)$$

```

<table>
  <mlabeledtr>
    <td>
      <mtext> (2.1) </mtext>
    </td>
    <td>
      <mrow>
        <mi>A</mi>
        <mo>=</mo>
        <mi>b</mi>
        <mo>+</mo>
        <mn>c</mn>
      </mrow>
    </td>
  </mlabeledtr>
</table>

```

The pseudo-code for translating mlabeledtr:

```

<xsl:template match=" mlabeledtr ">
  Get configuration parameters from the parent node.
  Add HTML <tr> tag for adding row.
  {
    Matches the child element except for the first one with configuration
    parameters.

    Match the first child element<!-- default is to put label at the far right -->
    {
      Add HTML <td> tag with align = right
      {
        Matches the child with configuration parameters.
      }
    }
  }
</xsl:template>

```

maligngroup

This element is especially created to use with the nested expression structure of layout schemata such as mtable, mrow, mphantom, etc. It basically allow the user to specify the alignment of each sub element at the top or parent element level.

Some of the group-alignment values are specified using the following syntax rules such as:

group-alignment := left | right | center | decimalpoint

group-alignment-list := group-alignment +

group-alignment-list-list := ('{' group-alignment-list '}') +

The main idea for this translation is to map or pass each setting value to the right `maligngroup` child according to the ordering of the value listed at the parent level of `maligngroup`.

malignmark

`malignmark` has one attribute, `edge`, which specifies whether the alignment point will be found on the left or right edge of some element or character.

Example: 10.55

122.35⁺

```
<table>
  <tr>
    <malignmark edge= 'right'>
      <td>
        <mn> 10.55 </mn>
      </td>
    </malignmark>
    <td>
      </td>
  </tr>
  <tr>
    <td>
      </td>
    <malignmark edge= 'left'>
      <td>
        <mo> + </mo>
      </td>
    </malignmark>
  </tr>
  <tr>
    <malignmark edge= 'right'>
      <td>
        <mn>122.35</mn>
      </td>
    </malignmark>
```

```
<td>
</td>
</tr>
</table>
```

The basic idea for translating `malignmark` element is simple, because our translation design is already a table based. We can create an alignment parameter and adjust the value according to `malignmark` attribute, and then send this parameter down to the children level for using with the alignment of its column, table, row, etc.

Actions

maction

This element simply allows the user to add the HTML capability to render the mathematical expression.

Example: $x+y$

```
<maction actiontype="highlight" my:color="red" my:background="yellow">
<mi>x</mi>
<mo>+</mo>
<mi>y</mi>
</maction>
```

The translating idea for `maction` is to create special parameters to render this special characteristic according to the `maction` attribute and then pass them down to its children level.

5) Deployment

I set up an experiment to compare the size of the result files that generated from the same document; one using the conversion program, TtH, from a LaTeX file to a graphic file in JPEG format, and another one using this project XSLT translation. The results show that for small number of mathematical expression lines, both result files have similar size, which is only a few KB. However, in the large number of mathematical expression lines, the size result file from the TtH software are much bigger than the mathematical document generated using this project translation. The file size comparison is shown below:

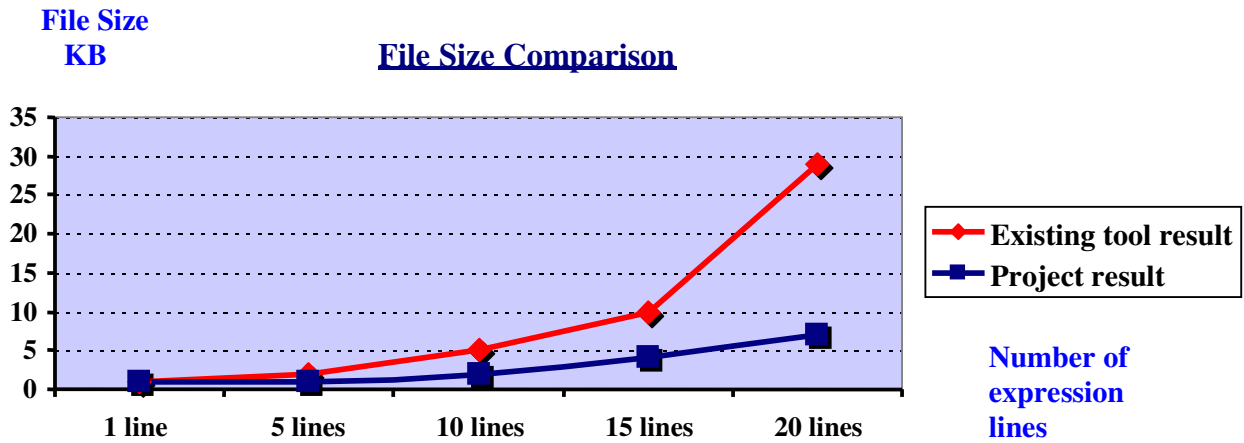


Figure 1: File size comparison chart

6) Conclusion

The desire to display mathematical expressions on the Web has been around for more than a decade. Many mechanisms previously proposed are still not adequate. This project proposes a breakthrough technique to display math on the Web without any hassle by transforming a XML file containing MathML to a XML file containing HTML and VML via a XSLT stylesheet transformation. Using this approach, mathematical expressions can be rendered on the Web without using any type of plugins. Furthermore, it takes a shorter time to view the result than the conventional way that uses image file such as GIF or JPEG, and also gives a better rendering result than those displaying the result using plain HTML.

This project has demonstrated the capabilities of XSLT in translating one language to another, and has also suggested a simplifying implementation strategy for translating MathML into another XML document that is viewable by popular browsers without using any types of plugins. Our experience with XSLT has shown that there are some limitations in terms of XSLT language that prevent us from implementing a full feature of MathML translation according to W3C recommendations for MathML version 2.0 as described in the design section. However, this project has accomplished most of the tags' translations with a reasonably satisfying quality of result rendering.

6) Bibliography:

- [ABS00] S. Abiteboul, P. Buneman, and D. Suciu. “*Data on the Web: From Relations to Semistructured Data and XML.*” Morgan Kaufmann Publishers, 2000.
- [BBMS99] Greg J. Badros, Alan Borning, Kim Marriott, and Peter Stuckey.
“Constraint cascading style sheets for the web.” *In Proceedings of the 1999 ACM Conference on User Interface Software and Technology.* pp.73-82, 1999.
- [BK02] F. Bry and M. Kraus. “Adaptive Hypermedia made simple using HTML/XML Style Sheet Selectors.” *In 2nd Int. Conf. on Adaptive Hypermedia and Adaptive Web Based Systems,* 2002.
- [D96] All about LaTeX2HTML. Nick Drakos. Retrieved January 20, 2002
from <http://cbl.leeds.ac.uk/nikos/tex2html/doc/latex2html/latex2html.html>.
1996.
- [D01] Nick Drakos. (2001) “All about LaTeX2HTML.” Retrieved January 31, 2002
from <http://cbl.leeds.ac.uk/nikos/tex2html/doc/latex2html/latex2html.html>.
[1996.](#)
- [D02] Math on the Web. Design Science. <http://www.dessci.com/webmath/>
- [ECMA99] Standard ECMA-262 ECMAScript Language Specification 3rd ed.
Retrieved January 31, 2002 from <http://www.ecma.ch/ecma1/stand/ecma-262.htm>. [ECMA. 1999.](#)
- [GR95] Goosens M., and Rahtz S. “From LaTeX to HTML and back.” *TUGBOAT.*
1995.
- [H01] TTH: the TEX to HTML translator. Ian Hutchison. . 2001.

- [K84] Knuth D.E. "The TeXbook. Computer and Typesetting, Vol. B." Reading MA. 1984.
- [KD99] Helmut Kopka and Patrick W. Daly. "A Guide to LaTeX 3rd Ed." Addison-Wesley. 1999.
- [M02] The HeVeA Home Page. Retrieved January 31, 2002 from <http://pauillac.inria.fr/~maranget/hevea/> Luc Maranget. 2002.
- [OHRE97] Van Ossenbruggen, J., Hardman, L. Rutledge, L., and Eliëns, A. "Style Sheet Supportfor Hypermedia Documents". *Proceedings of ACM Hypertext 97*. pp. 216-217, 1997.
- [W99a] WANG, P. S. "Design and Protocol for Internet Accessible Mathematical Computation." *In Proc. ISSAC'99, ACM Press*. pp. 291-298, 1999.
- [W99b] P. Wadler. "A Formal Semantics of Patterns in XSLT." *In Proceeding of the Conference for Markup Technologies*, 1999.
- [W3C00a]W3C (2000) "Extensible Markup Language (XML)." Retrieved January 20, 2002 from <http://www.w3.org/XML>.
- [W3C00b]W3C.(2000) Cascading Style Sheets, level 2 CSS2 Specification. Retrieved January 20, 2002 from <http://www.w3.org/TR/REC-CSS2>.
- [W3C01a]W3C.(2001) "W3C's Math Home Page." Retrieved January 20, 2002 from <http://www.w3.org/Math/>.
- [W3C01b]W3C.(2001) "VML - the Vector Markup Language." Retrieved January 20, 2002 from <http://www.w3.org/TR/NOTE-VML>.
- [W3C01c] Scalable Vector Graphics (SVG) Specification 1.0. Retrieved January 20, 2002 from <http://www.w3.org/TR/SVG/>. W3C.
- [W3C01d] W3C.(2001) "XSL Transformations (XSLT) Version 1.0." Retrieved January 20, 2002 from <http://www.w3.org/TR/xslt>.