MathML Formatting with TEX Rules and TEX Fonts*

Luca Padovani

University of Bologna, Department of Computer Science
Mura Anteo Zamboni, 7
40127 Bologna,
ITALY
email: lpadovan@cs.unibo.it
http://www.cs.unibo.it/~lpadovan/

Abstract

In this paper we investigate the architecture of a MathML formatting engine based on an abstraction of the TEX box primitives. This engine is carefully designed so that the TEX-dependent formatting rules are isolated from the independent ones and is capable of achieving a TEX-comparable formatting quality when used in conjunction with TEX fonts. We show how the formatting rules presented in Appendix G of the TEXbook can be easily adapted for MathML formatting, and how the semantically-rich MathML markup simplifies the rules themselves.

Introduction

The Mathematical Markup Language (MathML [4]) is an XML application for encoding mathematical formulae. It has two distinct sets of tags: the *presentation* tags which are used to encode what a formula looks like, and the *content* tags which are used to encode the "meaning" of a formula. The TEX macros for math typesetting either represent operators or identifiers that cannot be typed directly on the keyboard, or they implement the most common layout schemata for the mathematics. In the context of MathML presentation, the TEX macros of the first kind correspond to Unicode [1, 2] characters, hence they do not have dedicated markup. Macros of the second kind correspond very closely to MathML presentation tags, as Table 1 shows. We can thus say that TEX and MathML presentation encode mathematical formulae at the same level.

Indeed, the issue of formatting MathML documents using TEX can be addressed simply by providing a transformation from the MathML markup to the TEX/LATEX markup,[2] and using a TEX implementation for the actual rendering. This approach, however, is far from being all-satisfactory. For example, the whole rendering process is only feasible if no interactivity is required, that is, if the rendering can be performed off-line as a batch procedure on a static medium (the paper, an image). Also, the need for the large complex TEX system whose overall purpose goes far beyond math formatting is probably unreasonable. Furthermore, not every MathML document can be faithfully converted into the corresponding TEX markup—the conversion is more a good approximation than is an accurate rendering.

An alternative approach is to implement a self-contained MathML formatter that is also capable of using TEX fonts. However, the use of TEX fonts at this level requires a good understanding of the conventions adopted for the metrics of the glyphs they provide and the extra parameters they have. The TEX system relies on this knowledge in order to achieve a very high formatting quality. On the other side, MathML is a language for publishing and communicating mathematics on the Web, hence its rather high-level formatting semantics do not make any special assumptions on the graphic capabilities of the environment in which it is rendered, no matter whether it is a system for quality typesetting on the paper, a regular computer, or a hand-held device. In this paper we investigate how to carefully design a MathML formatter so to separate what is TEX-specific from what is common in every environment. We also show how the semantically rich MathML relieves the author from explicitly tweaking the markup, which is a potentially dangerous operation

* This work has been supported by the European Project IST-2001-33562 MoWGLI.

[2] Examples of such translation tools are the one developed at the Ontario Research Centre for Computer Algebra [8], and the XSLT stylesheets by Vasil I. Yaroshevich http://www.raleigh.ru/MathML/mmltex/.

Layout Schemata	TEX	MathML
Identifiers	a,x,\sin,...	mi
Numbers	0,1,2,...	mn
Operators	+,(,\oint,...	mo
Grouping	{ }	mrow
Fractions	\over, \atop	mfrac
Radicals	\sqrt, \root \of	msqrt, mroot
Scripts	_, ^	msub, msup, msubsup, mmultiscripts
Stacked expressions, lines and braces above and below formulae	\buildrel, \underline, \overline, \underbrace, \overbrace	munder, mover, munderover
Matrices, tables	\matrix, \cases	mtable

Table 1: Correspondence of TEX and MathML layout schemata

Name	Default	Description
size	inherited	font size
scriptLevel	0	number of nested scripts
minSize	8 *pt*	minimum font size that a script can be reduced to
displayStyle	inherited	true if formulae must be formatted in display mode
sizeMult	0.71	amount by which the font size is multiplied when the script level is increased by 1
stretchWidth	undefined	horizontal extent the operator is asked to stretch
stretchHeight	undefined	vertical extent (above the baseline) the operator is asked to stretch
stretchDepth	undefined	vertical extent (below the baseline) the operator is asked to stretch

Table 2: Properties of the MathML formatting context.

in the context of MathML and, more generally, of documents to be published on the Web, as there is no guarantee that the tweak will work successfully on a different formatter from the one the author is using.

We have chosen to proceed with our investigation by giving illustrations instead of exhaustive explanations, leaving to the reader the generalization of the cases considered here. Consequently, some knowledge of MathML and its basic concepts is highly recommended. We have assumed that the reader is also familiar with the TEX formatting rules for mathematics and other TEX concepts described in Appendix G of The TEXbook [5].

MathML formatting

By "formatting a formula" we mean the process that transforms the formula encoded in some markup language into a lower-level representation that conveys information about the needed glyphs, their size and their relative position.

Math formatting is always done with respect to a *formatting context* which defines, at least, (a) the (relative) font size at which formatting is occurring; (b) the scripting level—that is the number of nested scripts at which formatting is occurring; (c) whether the formula is formatted at display level (in a paragraph of its own) or inline. We define the MathML formatting context as a structure with a number of named *fields*. We use the "dot notation" to select a particular field in a formatting context, thus we will write $C.size$ to denote the value of the field *size* in the context C. Table 2 shows the main fields of a MathML formatting context.[3] A TEXnician will recognize that the MathML formatting context is just a generalization of the TEX notion of "style." In fact, the development of MathML has been influenced by TEX in many ways.

Because of the structured nature of MathML documents, MathML formatting can be expressed as a recursive function. Say we have a formula encoded in MathML like <t> $X_1 \cdots X_n$ </t> where t is the *type* of the root MathML element (ranging over mrow, mfrac, and so on) and the X_i its children, then the formatting of the formula in a given formatting context C, notation $[\![$<t> $X_1 \cdots X_n$ </t>$]\!]_C$, can be expressed as a proper combination, or re-arrangement, of the formatted children X_i each in its own formatting context C_i:

$$[\![\texttt{<}t\texttt{>}\ X_1 \cdots X_n\ \texttt{</}t\texttt{>}]\!]_C = \mathbf{f}_t(C', [\![X_1]\!]_{C_1}, \ldots, [\![X_n]\!]_{C_n})$$

[3] There can be many different definitions of a MathML formatting context. The one given here is not complete, but it suffices for the purposes of the paper.

Similarly, formatting of MathML token elements can also be expressed as a function of the current formatting context and of the Unicode characters $c_1, \ldots, c_n$ that the token is made of:

$$[\![\texttt{<}t\texttt{>}\ c_1 \cdots c_n\ \texttt{</}t\texttt{>}]\!]_C = \mathbf{f}_t(C', c_1 \cdots c_n)$$

In both these cases, the type t of the element being formatted and the value of attributes may affect the formatting context C and change it into a different context C'. For instance, formatting of a token element that explicitly sets the current font size (`mathsize` attribute) will format its characters in a context C' in which the field $C'.size$ has been updated accordingly. The complete set of rules for updating the formatting context C are described in detail within the MathML specification, and they basically follow the rules for style update of TEX (a concrete example will be given later).

As is described in Appendix G of The TEXbook, TEX scans and processes a logical representation of a formula (a *math list*) consisting of items, converting it into a physical representation (a horizontal list) made up of regular boxes. Items in the math list can be of different types, and in many cases they directly correspond to basic math layout schemata (like Rad atoms for radicals, Acc atoms for accents, or generalized fractions). For every item type, Appendix G defines one or more rules describing in detail how the item is converted in one or more boxes, ultimately defining how TEX formats mathematical formulae. Given the strict correspondence between TEX commands for math typesetting (hence math list items) and MathML presentation elements, the same set of rules can be easily adapted for the definition of the $\mathbf{f}_t$ functions.

In the following sections we consider in some more detail the formatting rules for the main categories of MathML elements and show how they map on TEX rules.

Groups. Math lists result from typesetting in math mode. The content of a math list is typically formatted on a single horizontal line, with all the items aligned on their baseline. In this respect math list are close to the `mrow` element. However, `mrow` is also fundamental for the following purposes:

- all stretchable operators within the same `mrow` element should vertically stretch so as to have the same height plus depth, unless they are constrained in some way either by the markup or by some limitations of the environment;
- `mrow` is the main MathML element enabling automatic line-breaking of long formulae, when they exceed the available horizontal space for formatting. Conversely, TEX grouping operators `{ }` freeze a sub-formula preventing any line-break in it.

Stretching of operators is done by looking at the bounding box of the child elements, once they have been formatted, and then passing adequate information through the formatting context, the idea being that operators have to take care of stretching themselves. This generalized treatment of stretchy operators relieves all the other MathML elements from taking into account vertical stretching rules. In particular, rule **13** (large operators) only applies when formatting stretchable operators (see the discussion on tokens that follows), and rule **15e** (fractions with delimiters) is never necessary because if a fraction must have delimiters, they must be explicitly encoded inside an `mrow` element along with the delimited fraction.

Automatic line-breaking is only affected by the bounding box of the formatted child elements, possibly requiring re-formatting of all or some of them. This can be considered a higher-level formatting issue which does not involve low-level TEX rules.

Tokens. Tokens are the basic building blocks of every mathematical formula. In MathML, tokens are the only elements allowed to have actual text as content. The most important token types are `mi` for identifiers, `mn` for numbers, `mo` for operators. The first two correspond roughly to Ord atoms in a math list, whereas the last one is refined in TEX into different atoms, Op, Bin, Rel, Open, Close, or Punct. The most remarkable difference is that in TEX those atoms can be made of exactly one character, whereas MathML tokens are made of arbitrary Unicode strings. From the point of view of formatting, the more general scheme adopted by MathML does not pose any additional problem, though, and it actually simplifies the encoding of non-strictly mathematical documents in which identifiers and operators whose name is longer than one character are frequent.

Note also that the fine-grained classification of operators is needed in TEX for mainly two reasons: (1) computing the right amount of space around operators; (2) helping the automatic line-breaking algorithm with hints on where the formula can be broken. In particular there is a distinction between unary operators (Op) and binary operators (Bin) as they typically have different spacing rules. In properly grouped MathML markup there is no need for distinguishing unary (prefix and postfix) from binary (infix) operators, as their *form* can be inferred from their position in the enclosing `mrow` element.

Rule **14** does not apply anymore in general for if two characters are marked up in different tokens they should never be kerned or merged into a ligature. This behavior is rather part of the formatting semantics of the token itself.

Operators are by far the most complicated tokens to format, for they may have to stretch vertically or horizontally and are affected by a large number of MathML attributes. Assuming that information about stretching is propagated in the formatting context (*C.stretchWidth*, *C.stretchHeight*, and *C.stretchDepth*), operators can use rule **19** for determining the exact extent they should span, and rule **13** when they must be formatted in the large form.

Accents. MathML provides multiple ways for encoding an "accent," depending on whether the accent is meant to be syntactical or semantical. A syntactical accent is simply part of a name, it has no mathematical meaning. Although it is very rare in mathematics to have identifiers with accents, the use of explicit markup allows MathML to disambiguate the two cases. Syntactical accents are used following the Unicode rules for combining characters. Hence, a MathML identifier like

```
<mi> a&#x0307; </mi>
```

is typeset as $\dot{a}$ (the Unicode character `U+0307` represents the "combining dot above" ◌̇).

A semantical accent usually denotes an operation, like first-order derivative in case of the "combining dot above." As such it is marked up as

```
<mover accent="true">
  <mi> a </mi>
  <mo> &#x0307; </mo>
</mover>
```

even though the formatted result is likely to be the same as in the previous case. Note how the accent is explicitly marked up as an operator inside an `mo` element. Rule **12** (accents) can handle both accents combined with a single character and wide accents combined with arbitrary subexpressions.

Although they cannot be considered proper accents, horizontal lines extending above or under a formula are treated in MathML uniformly with all the other operators. In particular, the MathML markup corresponding to `\underline{a}` is

```
<munder>
  <mi> a </mi>
  <mo> &#x0332; </mo>
</munder>
```

where `U+0332` is Unicode combining horizontal line below. The horizontal stretching rules of MathML operators require the `U+0332` character to stretch to the width of the base subformula. The symmetric situation (`\overline`) is handled similarly, except that the Unicode character to be used is `U+0305`. These two cases are handled by rules **9** and **10**.

Radicals. Formatting of root symbols deriving from `msqrt` and `mroot` elements is handled by rule **11** concerning Rad atoms.

Fractions. The `mfrac` element governs encoding of fractions. In TEX, alignment of the numerator and the denominator can be determined by the use of `\hfill`. In MathML, it is affected by the value of the attributes `numalign` and `denomalign`. TEX rules for formatting fractions are those from **15** through **15d**. TEX provides different commands for typesetting vertical material in a fraction-like manner, depending on whether one wants a fraction bar (`\over`, `\above`) or not (`\atop`). MathML handles all such cases by setting the `linethickness` attribute.

Scripts. Scripts are handled in MathML by the `msub`, `msup`, `msubsup`, `mmultiscripts` and sometimes `munder`, `mover`, and `munderover`. The reason why `munder`, `mover`, and `munderover` elements have to do with scripting is that they implement a mechanism similar to that determined by the `\limits`, `\nolimits`, `\displaylimits` commands in TEX.

TEX rules governing the placement of scripts are those from **18** through **18f**. Note that scripts in TEX are represented as possibly empty fields on any atom, whereas they are uniformly marked up with elements in MathML.

Tables. MathML markup for tables does not involve any specific font dependency as it basically is a higher-level formatting problem, if compared to formatting of the other elements.

Dealing with TEX dependencies

If we define an *environment* as the combination of available fonts, graphic capabilities of the output medium and user requirements, TEX rules for formatting make strong assumptions that are hard to generalize to environments other than typesetting math on the paper using a family of TEX fonts [7]. We can summarize TEX dependencies as follows:

- non-standard font metrics (like the thickness of the horizontal line in a radical which is computed from the height of the radical symbol, or the use of width and italic correction for the placement of scripts);
- non-standard kerning information (like using `\skewchar` for determining the horizontal displacement of accents);
- font-related quantities (the parameters σ_i and ξ_j in Appendix G) whose value cannot be otherwise inferred or computed in general;
- use of boxes of "black ink" (rules) for drawing fraction lines, root lines, joining segments in horizontal braces, formatting of Unicode characters `U+0305` and `U+0332`;
- builtin TEX constants (see `\delimiterfactor` and `\delimitershortfall` for stretchy operators, `\nulldelimiterspace`).

It is not feasible for any environment in which we might be interested to format MathML markup to provide the same set of parameters, or to address specific formatting issues the same way TEX does. Neither is it feasible to assume that the set of parameters we have considered is a superset of all the possible parameters that may ever affect formatting of mathematical formulae. Nonetheless the list of operations involving dependencies (formatting of tokens, radicals, scripts, fractions, accents) will be exactly the same in systems other than TEX. This amounts to saying that the definition of each $\mathbf{f}_t$ can be split up into two components: a $\mathbf{g}_t$ component that does not depend on anything that is TEX specific, and a $\mathbf{h}_t$ component that is strictly dependent on TEX. By doing so we automatically identify two main parts of the MathML formatter: the set of functions $\mathbf{g}_t$ which defines the *formatting engine*—that part which takes care of all TEX-independent aspects of the formatting; the set of functions $\mathbf{h}_t$ which defines the TEX *device for mathematics*—that part which deals with anything which is (or may be) dependent on TEX fonts or TEX formatting rules.

Since the set of $\mathbf{f}_t$ is finite and agreed upon *a priori* (the set of math layout schemata is fixed and relatively stable, being the result of centuries of slow evolution and convergence to modern notation), the interface to the TEX dependent parts is also fixed. This way when the environment changes—say when we move to a different family of fonts that does not adopt the TEX conventions, we need only to re-instantiate the $\mathbf{h}_t$'s with those that are customized to this new environment but with the same agreed interface, while sharing the same set of $\mathbf{g}_t$'s.

Figure 1 summarizes graphically the entities involved in formatting a fraction element, in particular the separation of the $\mathbf{g}_{\texttt{mfrac}}$ and $\mathbf{h}_{\texttt{mfrac}}$ components.

Of course the proposed modular organization of the formatter makes sense only if the two sets of functions are both performing non-empty and non-trivial tasks. But it is clear from Appendix G of The TEXbook [5] and the intricacies of math formatting detailed therein that the TEX dependent part is non-trivial. As for the formatting engine part, it has the following list of non-trivial responsibilities that we cannot elaborate further here for the sake of brevity: the construction of a data structure (typically a tree) which is suitable for formatting purposes; the implementation of the MathML mechanism for attribute inheritance and evaluation; the computation of updated formatting contexts; the algorithm for automatic line-breaking of long formulae; the algorithm for table layout.

An area model for MathML

The formatting functions $\mathbf{f}_t$ take objects as arguments and produce a new object as a result. Such objects, which we will call *areas* from now on, are a low-level representation of formatted formulae, and constitute what we define as the *area model* of the TEX device for mathematics (or, in general, any other instance of it). From the point of view of the formatting engine, we note that the only thing that matters is the ability of computing an area's bounding box, which is essential for updating the context with information about stretchable operators, for the automatic table layout algorithm and for the line-breaking algorithm.

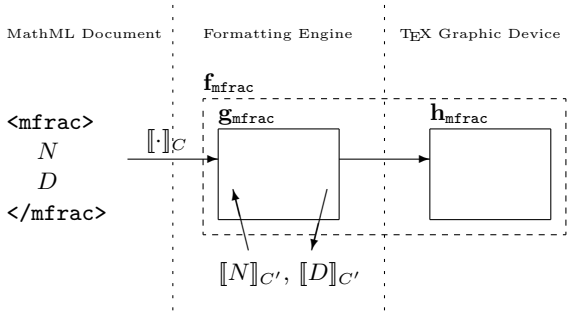

Figure 1: Modularization of the formatting function for the `mfrac` element.

Area	TEX box	Description
$G[\cdot]$		Glyph
K_n	`\kern`	Kern with value n. The kern is horizontal or vertical depending on the container it is in
F	`\hfill \vfill`	Filler area
R_n	`\hrule \vrule`	Filler rule of thickness n
$S_n[\alpha]$	`\raisebox \lowerbox`	Shift α's baseline by n
$H[\alpha_1, \ldots, \alpha_n]$	`\hbox`	Horizontal group of areas $\alpha_1,\ldots,\alpha_n$
$V_k[\alpha_1, \ldots, \alpha_n]$	`\vbox`	Vertical group of areas $\alpha_1,\ldots,\alpha_n$, where α_k is the reference area that determines the baseline

Table 3: Area types and their rendering semantics.

We summarize this by saying that areas are *opaque* to the formatting engine. From the point of view of the $\mathbf{h}_t$ functions, however, areas must convey more information. Many of the formatting rules presented in Appendix G have dependencies on the kind of areas being combined (for example, whether they are simple glyphs or arbitrary formatted sub-formulae), and also on the actual shape of the glyphs the areas are made of. We summarize this by saying that areas must be *transparent* to the TEX device for mathematics. The neat separation of views (opaque and transparent) of the area model is crucial in the design of a modular architecture in that it relegates areas to the TEX dependent part of the formatter, the only requirement for them being that of exporting a very limited set of operations (in fact, the computation of the bounding box).

In the specific case of formatting using TEX rules and TEX fonts, the area model can be naturally synthesized as a subset of TEX boxes. Table 3 introduces an abstract notation for the most common area types needed for math formatting, along with their corresponding TEX box or primitive command that achieves the same (or a similar) formatting. The reason for not using TEX primitive boxes directly is that the very same abstract model can be implemented in a sensitive way depending on the environment, whereas the TEX box model is obviously tied to TEX. For example, when formatting a MathML document that is meant to be interactive, the resulting areas may carry information about selections, or may have backward pointers towards the MathML elements that generated them. Such information is clearly unnecessary when formatting a static document for printing.

Example (`mfrac` formatting). We conclude this section by showing a complete example of formatting function, along with the resulting area object.

Let's suppose we are to format a subformula of the form `<mfrac>` N D `</mfrac>` representing the fraction

$$\frac{N}{D}$$

where we may assume, for the sake of generality, that N and D are metavariables standing for arbitrary subexpressions rather than actual identifiers. Then we have

$$[\![\texttt{<mfrac>}\ N\ D\ \texttt{</mfrac>}]\!]_C = \mathbf{f}_{\texttt{mfrac}}(C, \alpha, \beta)$$

where

$$\alpha = [\![N]\!]_{C'} \qquad \beta = [\![D]\!]_{C'}$$

and C' is such that if $C.displayStyle$ = true then $C'.displayStyle$ = false and all the other fields are the same as in C, whereas if $C.displayStyle$ = false then $C'.scriptLevel = C.scriptLevel + 1$, $C'.size = \max(C.minSize, C.size \times C.sizeMult)$ and the other fields are the same as in C.

Depending on the value of the `numalign` and `denomalign` attributes the numerator and the denominator may be aligned to the left, to the right, or may be centered (this is the default). Assuming they are centered, $\mathbf{f}_{\texttt{mfrac}}$ is defined as

$$\begin{aligned} & \mathbf{f}_{\texttt{mfrac}}(C, \alpha, \beta) \\ = \; & \mathbf{h}_{\texttt{mfrac}}(C, H[F, \alpha, F], H[F, \beta, F]) \\ = \; & S_a[V_3[H[F, \beta, F], K_d, R_h, K_n, H[F, \alpha, F]]] \end{aligned}$$

which is to be read as follows: the fraction is made of (from bottom to top) the centered denominator $H[F, \beta, F]$, a kern d, the horizontal bar of thickness h, a kern n, the centered numerator $H[F, \alpha, F]$. The reference point of the whole vertical area V coincides with the reference point of its third child (the horizontal bar). Then, the whole fraction is shifted up by a so that the horizontal bar is aligned with the axis of the expression. The quantities a, d, h, and n are computed by $\mathbf{h}_{\texttt{mfrac}}$ following rules **15-15d**.

Note how the process is clearly split into a TEX independent part (computation of updated formatting contexts, alignment of numerator and denominator) and a TEX dependent one (exact positioning of the subparts).

How semantics helps formatting

When we speak of "semantics" in the context of MathML presentation markup, we refer not only to the presentation tags but also to the following characteristics:

- The use of explicitly encoded, although invisible, operators such as "invisible multiplication" and "function application;"
- The use of a more rigid encoding of mathematical formulae, in particular the proper grouping rules. In practice this amounts at allowing within the same `mrow` element at most one kind of operator, with only a few exceptions (pluses and minuses within the same `mrow` are not considered a violation to the proper grouping rule);
- The use of a dictionary that customizes the basic properties of known operators, such as their stretchability, whether they are delimiters, fences, or other kinds of operators, the amount of space that should normally be put around them.

The presence of semantics in the MathML encoding of a mathematical expression has often been associated with the ability of reconstructing a more semantically-oriented representation of the same expression. However, such information can also be exploited for formatting purposes as it allows the formatter to apply context-sensitive rules that are logically related to the semantics of the entity being formatted. This eliminates the need for strange formatting rules, special cases, and other oddities that abound in the formatting rules of TEX markup, and it also permits a more general and effective formatting than TEX allows. In the sections that follows we consider three specific examples.

Invisible operators are explicit. The use of explicit markup for invisible operators has a noticeable impact on the formatter and the way it operates. As a concrete example lets consider a typical trigonometric function, say sine, for which an explicit TEX macro has been designed. Depending on the author's taste or needs, there are two different ways to typeset the sine of x:

$$\sin x \quad \text{or} \quad \sin(x)$$

both of which are encoded using the `\sin` macro. A close look at the formatted formulae will reveal that in $\sin x$ there is a little space between the function name and the argument, whereas there is no space in $\sin(x)$. Although this is indeed very natural, and yields a nicely formatted formula, the hidden mechanism that makes this happen is anything but straightforward. The definition of the `\sin` macro is something like the following:

```
\def\sin{\mathop{\rm sin}}
```

The key point is the definition of `\sin` as an operator. By TEX spacing rules, an operator (Op atom) followed by a variable (Ord atom) gets some space after it, but the same operator, when followed by a delimiter (Open atom), gets no space at all.

In MathML the same expressions would be encoded as:

```
<mrow>
  <mi>sin</mi>
  <mo>&ApplyFunction;</mo>
  <mi>x</mi>
</mrow>
```

or as:

```
<mrow>
  <mi>sin</mi>
  <mo>&ApplyFunction;</mo>
  <mrow>
    <mo>(</mo> <mi>x</mi> <mo>)</mo>
  </mrow>
</mrow>
```

respectively. The fact that we have an explicit operator, function application, between the function and its argument allows the formatter to do something context-sensitive along the following lines: whenever formatting of `ApplyFunction` operator is requested, look at the next element. If it is an `mrow` whose first child is a delimiter then render `ApplyFunction` as 0 width space. Otherwise render it as some suitable constant space. The difference with TEX is that this behavior is explicitly associated with formatting of the `ApplyFunction` character, and is not part of a more general (but also less clear) scheme for spacing math atoms. Finally, note that TEX's mechanism relies on the `\sin` function being defined as an "operator," whereas `sin` is correctly marked up as an identifier (for the sine function) in MathML markup.

Without entering the same level of detail, the reader can easily verify that a similar thing happens with invisible multiplication, which is in fact invisible in TEX markup, whereas it is explicitly encoded as the `InvisibleTimes` operator in MathML. In the case of adjacent fractions, just to mention one specific case, null delimiters with non-null width are accurately placed so that the fraction bars do not join together. Correct spacing between math Ord atoms is also ensured by an accurate use of font metric information, italic correction in particular, which guarantees a very high quality of the formatted formula, but contributing in making TEX fonts and TEX formatting rules mutually dependent.

Opening and closing delimiters. TEX distinguishes delimiters as opening and closing depending on their name. (, [, { are examples of opening delimiters, ⌋, ⌉, ⟩ are examples of closing delimiters. The distinction is carried on at the level of atoms, where the delimiters are represented by either Open or Close atoms. In a properly grouped MathML expression the distinction is made depending on the *position* of the operator rather than its name. In fact, a properly grouped expression must be a `mrow` element whose first and last children are the opening and closing delimiter, respectively, and the middle child is the body of the expression. The operator dictionary that determines the default value of operator properties, spacing in particular, is addressed by both the operator's name and its form (one of prefix, infix, or postfix). Hence, the use of properly grouped markup combined with an operator dictionary provides for greater generality and flexibility in a formatter for MathML markup. As a noticeable side effect, it also disambiguates those cases in which the opening and closing delimiters are equal (think of | or ‖), which must be carefully treated in TEX markup in order to get the spacing right.[4]

The strange case of the solidus symbol. Inline division is typically represented by the / symbol to be placed very close to its operands. In TEX there is this oddity that the / symbol is *not* treated as an operator Op, but rather as an ordinary symbol Ord, so that it gets no space around during the second phase of formula formatting (rule **20**). The visual effect is that of rendering as $1/2$ rather than $1 / 2$. In MathML formatting this trick is no longer necessary, and the / can be naturally encoded as `<mo>/</mo>`, for the spacing around it can be controlled by the `lspace` and `rspace` attributes accepted by every `mo` element (their default values can be specified in the MathML operator dictionary).

Conclusions

As MathML is relatively similar to TEX, at least at some abstract level, most of the TEX formatting rules can be easily adapted and used for MathML formatting when TEX fonts are available. However, doing so in a modular and adaptable way, without necessarily committing to TEX fonts, is a more delicate problem.

In this paper we have surveyed a number of issues and their possible solutions, ultimately depicting the architecture of a formatter for MathML markup which is capable of exploiting all of TEX's finest rules

[4] TEX also provides for a `\mid` operator which should be used for | when it stands as the separator in comprehensive notation for sets. This case is also simplified in MathML markup, as in properly grouped markup the | operator would be correctly treated as an "infix" operator.

for math typesetting without being tied at the same time to the TEX fonts. Thanks to the structurally and semantically rich MathML markup, the formatter also succeeds in cases that cannot be handled in a general way by the TEX formatting rules without the help of the author. This aspect is particularly relevant because, if the desired rendering is not achieved by the TEX rules, the author can tweak a TEX formula and still be sure that the formula will be rendered the same way on every system running TEX. On the other hand, since MathML formatters are not tied to a set of fixed formatting rules, tweaking the MathML markup can potentially compromise an effective rendering of the formula.

The problem of being adaptable to the formatting environment is not just a matter of recognizing the available fonts and achieving the finest formatting with those fonts. TEX formatting rules assume that the formulae will be eventually printed on paper, or anyway displayed on a high resolution screen. There are contexts in which it is more convenient to display symbols differently, as to improve editing, interaction, or readability, especially in low-resolution display such as those used in hand-held devices.

The techniques that we have described in the paper have been successfully put into practice in two prototypes, a MathML formatting engine for a recognizer of hand-written mathematics in hand-held devices at the Ontario Research Centre for Computer Algebra [9], and the `gtkmathview` widget[5] at the University of Bologna. The latter tool, in particular, has been adopted by John Wiley & Sons, Inc., the publisher, for rendering mathematical formulae encoded in MathML markup achieving a quality comparable to that of TEX and using several families of TEX fonts.

In a broader perspective, the architecture we have designed and implemented allows applications to exploit context dependencies, instead of avoiding them. As the development of the two prototypes has shown, the efforts required for implementing the techniques are negligible when compared to the potential benefits.

References

[1] "The Unicode Standard", Version 3.0, Addison Wesley, 2000.

[2] "Unicode Standard Annex #28, Unicode 3.2", 2002. http://www.unicode.org/unicode/reports/tr28/

[3] "Extensible Markup Language (XML) Specification", Version 1.0, W3C Recommendation, 10 February 1998. http://www.w3.org/TR/REC-xml

[4] "Mathematical Markup Language (MathML) Version 2.0", W3C Recommendation, 21 February 2001. http://www.w3.org/TR/MathML2/

[5] D. E. Knuth, "The TEXbook", Addison-Wesley, Reading, MA, 1998.

[6] D. E. Knuth, "The METAFONTbook", Addison-Wesley, Reading, MA, 1994.

[7] U. Vieth, "Math typesetting in TEX: The good, the bad, the ugly", Proceedings of the EuroTEX Conference, 2001, The Netherlands.

[8] E. Smirnova, S. M. Watt, "MathML to TEX Conversion: Conserving high-level semantics", MathML International Conference 2002, http://www.mathmlconference.org/2002/presentations/smirnova/index.html

[9] L. Padovani, "A Standalone Rendering Engine for MathML", MathML International Conference, Chicago, IL, 2002. http://www.mathmlconference.org/2002/presentations/padovani/

[10] L. Padovani, "MathML Formatting", Ph.D. Thesis, Technical Report UBLCS-2003-03, Dept. Computer Science, Bologna, Italy, 2003.

[5] See http://helm.cs.unibo.it/mml-widget/. [10]