

# The Design of MathML-Content 2 Revision 2 and Implications for Unifying MathML and OpenMath

Andreas Strotmann

School of Public Health, University of Alberta, Edmonton, Alberta, Canada

**Abstract.** What was supposed to be the final major revision of MathML-Content was undertaken in the run-up to Revision 2 of MathML 2. While the current effort to update MathML to version 3 introduces the notion of a "strict" subset of MathML-Content as a major change, it appears to explicitly refrain from making any major changes to the results of that final revision in the "non-strict" part of MathML-Content.

In this paper I discuss the view of the structure of MathML-Content on which that final revision was based, and reasons why this particular view was adopted. While closely related to the currently available draft of "strict" MathML-Content, this view of what MathML-Content 2 "meant" differed in crucial ways from the current draft.

Adherence to the "original" view of MathML-Content may require more radical adjustments to OpenMath, and less radical adjustments to MathML-Content, than are currently being considered in an effort to unify the two.

## 1 Introduction

In the early summer of 2003, I conducted a systematic review of Content-MathML as defined in the then final draft of Edition 2 of MathML 2 [4]. In May 2003, the results of this review were submitted to the MathML developers mailing list as a series of long lists of suggested fixes accompanied by rationales [6–8], and a discussion on these proposals continued for several weeks, both on the `www-math` mailing list and, as the discussion grew more technical and involved, off the list with members of the MathML working group, primarily Stan Devitt.

As Stan noted in his first response to my review [5], my comments revolved around a deep structural analysis of the interplay between the various forms of specifying variable bindings and the various forms of specifying the domains that these variables range over in a piece of mathematics expressed as a Content-MathML expression. The goal of my review was to suggest ways of providing a coherent and consistent semantics for the numerous qualifiers and elements available in MathML 2.

In the current working draft of MathML 3 [3], the first major revamping of MathML since revision 2 of MathML 2, this goal, which was left implicit in the Recommendation itself even though it was openly discussed on the mailing list

[7], is now being made quite explicit by introducing and specifying the notion of a “strict” subset of MathML-Content. The idea, if I understand it correctly, is to specify a fairly minimal subset of MathML-Content which can express the full range of meanings that the rest of MathML-Content can express, but which does so systematically.

The current draft of “strict” MathML-Content appears to be heavily influenced by OpenMath 2 [1], rather than starting from the existing MathML-Content specification itself. It is therefore useful to take another look at what the underlying intentions were that originally lead to the definition of Content-MathML as it (still) stands, without trying to force it into the quite distinct conceptual framework of OpenMath.

## 2 Binding and Ranging in MathML 2 Revision 2

MathML 2 offers many different ways of expressing three fundamental concepts for structuring mathematical expressions: (function) application, (variable) binding, and the ranging of operators, functions or variables over “domains of application.” OpenMath 2, by contrast, only offers a single way of expressing the first of these two, and has to express the third via function application (usually, different argument patterns, or in the form of a domain of application assignment operator).

Application in MathML is expressed in the form of an **apply** element, while binding and ranging are both expressed in the form of qualifier elements that are children of other elements. There is a single variable binding qualifier, *bvar*, which may appear inside most MathML elements to define the scope of variables within their bodies. MathML offers quite a collection of qualifiers that specify the domain that operators or variables range over within an expression: *interval*, *lowlimit*, *uplimit*, *domainofapplication*, and *condition*. The latter restricts variable ranges through predicates (including predicates over variables bound in a sibling *bvar* qualifier), while the others define sets over which a variable ranges.

To make sense of all these qualifiers, both individually and in concert, and in order to reach a systematic and consistent understanding of them, we considered a number of principles that the final outcome of this review of Content-MathML should adhere to. In this paper, I attempt to reconstruct them from both public and private records of the discussion around this review.

The first principle regarding Content-MathML qualifiers that the review and revision were based on was as follows:

**Principle 1.** Binding and ranging qualifiers are orthogonal.

Principle 1 states that, in general, any form of expressing binding and any form of expressing ranging may be combined freely, and semantic equivalences between forms of expression in either class apply regardless of presence or absence of forms expressing the other class. This includes, in particular, that all these qualifiers are optional, again regardless of the presence or absence of qualifiers in the other class.

We therefore discuss the two classes of concepts separately, before returning to a discussion of the interrelationships between them.

## 2.1 Variable Bindings

Bound variables are basically expressed in the form of lists of child elements of the *bvar* qualifier embedded in either an **apply** or a **lambda** element. In both cases, the scope of the bound variables declared in the *bvar* qualifier is the body of the element of which the *bvar* qualifier is a child element. If the *bvar* qualifier is inside an **apply** element, the head of the **apply** element is an operator or quantifier.

**Principle 2.** The pattern **apply**(operator, *bvar*(vars), body) is semantically equivalent to the pattern **apply**(operator, **lambda**(*bvar*(vars), body)).<sup>1</sup>

Note that Principle 2 applies across the board, regardless of which other qualifiers are present within an **apply**. However, scoping rules governing sibling qualifiers determine whether or not these qualifiers should be children of the **apply** or of the **lambda** element in this equivalence – we will discuss this below.

## 2.2 Domains of Application

Ranging of variables, operators, quantifiers or functions over domains of application is expressible in a number of ways in MathML 2:

- intervals are specified as the *interval* qualifier or as one or both of the *lowlimit* and *uplimit* qualifiers.
- general sets are specified as *domainofapplication* qualifiers

**Principle 3.** Interval-specifying qualifiers are equivalent to an appropriate **interval** constructor embedded inside a *domainofapplication* qualifier.

Semantically, the contents of these set definition qualifiers is *outside* the scope of any sibling *bvar* variables. In Principle 2 above, they would therefore appear as children of the **apply** element on both sides of the equivalence.

By contrast, the *condition* qualifier is explicitly *within* the scope of any sibling *bvars*, and would thus appear as a child of the **lambda** element on the right hand side of the equivalence in Principle 2. It is meant to constrain the range of application by specifying conditions on these variables. Nevertheless, we can state a semantic equivalence as follows:

---

<sup>1</sup> For the sake of brevity, I will refrain from using XML syntax in this paper. Instead, the typographical conventions that I follow here for Content-MathML expression patterns are as follows: bold face is used for names of elements such as **apply** and **lambda**, italics are used for names of qualifiers such as *bvar* and *condition*, and regular type face is used to name parts of a Content-MathML expression pattern that are left unspecified.

**Principle 4.** The pattern `apply(a, bvar(vars), condition(P), body)` is semantically equivalent to the pattern

`apply(a, bvar(vars), domainofapplication(apply(set, bvar(vars), P)), body)`

Others have argued that, while Principle 4 may be correct in the sense of an abstract extensional semantics, this interpretation is problematic in a pragmatic sense, in that semantics-preserving transformations such as  $\alpha$ -conversion may now rename variables in one of the two distinct places where they appear but not the other, destroying the original identity link between them. As we will see in the next subsection, this connection is actually maintained semantically even if, pragmatically, it may appear broken.

Finally, we have:

**Principle 5.** Only one of these groups of qualifiers may be used in an element: *lowlimit/uplimit*, *interval*, *domainofapplication*, or *condition*.

The MathML Recommendation specifically advises against using *interval* or *lowlimit/uplimit* when *domainofapplication* is used in the same element; this principle is a generalization that was proposed verbatim in 2003.<sup>2</sup>

### 2.3 Interplay between Bound Variables and Domains of Application

When an `apply` element includes both bound variable and domain of application qualifiers, they are connected through the usual mathematical convention:

**Principle 6.** In the presence of a *bvar* sibling qualifier with  $n$  bound variables, a *domainofapplication* qualifier is expected to denote an  $n$ -dimensional space, with variables mapped to dimensions in the order of their appearance in the *bvar* qualifier.

In addition, as mentioned in the previous section, scoping rules state that *condition* qualifiers are within scope of a sibling *bvar* qualifier, and other domain of application qualifiers are out of scope for the variables listed in a sibling *bvar*.

### 2.4 Interplay between Application, Binding, and Ranging

**Principle 7.** The above principles apply to bound variables and to domains of application independent of the form or the meaning of the head of an `apply` element that they are children of.

In particular, the head of an `apply` may be a user-defined or a compound expression without affecting the equivalence principles stated in this section.

---

<sup>2</sup> It may make sense to allow both a domain-of-application type qualifier and a condition qualifier in the same element, however.

### 3 Example

The following are semantically equivalent according to the principles above:

- **apply**(**int**, *lowlimit*(0), *uplimit*(**pi**), **sin**)
- **apply**(**int**, *interval*(0, **pi**), **sin**)
- **apply**(**int**, *domainofapplication*(**apply**(**interval**, 0, **pi**)), **sin**)

The following are semantically equivalent as well, but the principles do not specify that they are equivalent to the previous set of examples:

- **apply**(**int**, *bvar*(*x*), *interval*(0, **pi**), **apply**(**sin**, *x*))
- **apply**(**int**, *bvar*(*x*), *domainofapplication*(**apply**(**interval**, 0, **pi**)), **apply**(**sin**, *x*))
- **apply**(**int**, *bvar*(*x*), *condition*(**apply**(**in**, *x*, **apply**(**interval**, 0, **pi**))), **apply**(**sin**, *x*))
- **apply**(**int**, *interval*(0, **pi**), **lambda**(*bvar*(*x*), **apply**(**sin**, *x*)))
- **apply**(**int**, **lambda**(*bvar*(*x*), *condition*(**apply**(**in**, *x*, **apply**(**interval**, 0, **pi**))), **apply**(**sin**, *x*))

Here is a minimalist example with no qualifiers whatsoever:

- **apply**(**int**, **sin**)

The following are equivalent:

- **apply**(**int**, *bvar*(*x*), **apply**(**sin**, *x*))
- **apply**(**int**, **lambda**(*bvar*(*x*), **apply**(**sin**, *x*)))

### 4 Derivation of a Strict MathML 2

Based on the principles listed in Section 2, we can derive a “strict” MathML 2 that mirrors these principles. However, since equivalence principles as listed mostly apply in both directions, there is no unique solution.

By applying all of the above equivalence principles, the most radical, and perhaps the cleanest, version of a strict MathML 2 would reduce its structural elements to **apply**, **lambda**, *bvar* only inside **lambda**, and *domainofapplication* as an optional element only of **apply**. This has the advantage of explicitly moving *domainofapplication* out of the scope of any *bvar* expression, thus clarifying a major source of confusion among users of MathML 2. It is a truly minimalist solution, a language theorist’s dream, with great potential for defining a corresponding clean formal semantics of the sort that was defined for OpenMath 1 in [10], only cleaner.

A less radical version of a strict MathML would have been closer to our vision back then, although the above, radical solution was explicitly maintained as an option throughout the revision process. In this version, **lambda** would be removed (applying Principle 2 in the opposite direction), leaving **apply** with optional *bvar* and *domainofapplication* children (at most one each), with the explicit rider that *domainofapplication* is out-of-scope for the *bvar* variables.

Finally, as a compromise between the vision behind MathML 2 and the current draft of Strict MathML 3, by not applying Principle 4, we could derive a strict MathML that would retain only **apply** with optional *bvar*, *domainofapplication*, and *condition* qualifiers, again with the explicit rider that *domainofapplication* is outside scope and *condition* within scope of *bvar*.

#### 4.1 Comparison with Draft Strict MathML 3

The current draft of a Strict MathML 3, by contrast, reduces its structural elements to **apply** with optional *bvar* qualifier (in which case the **apply** is to be renamed to **bind**, a complication that we will ignore for the moment) and an optional *condition* qualifier which is explicitly within scope of the sibling *bvar* variables.

This version of a strict MathML cannot be derived from MathML using the principles above, for several reasons.

For one, the equivalence principle between *domainofapplication* and *condition* qualifiers (Principle 4) is complete for *condition* qualifiers, but not for *domainofapplication* qualifiers, i.e., any *condition* qualifier is equivalent to a corresponding *domainofapplication* qualifier, albeit a somewhat complex one, but only *domainofapplication* qualifiers of a very special form are equivalent to *condition* qualifiers in that equivalence. This is not just a problem of the specific form that that principle takes, either:

**apply**(operator, *domainofapplication*(D), f)

is an expression that cannot be expressed using *condition* as it contains no variables and the arity of f is unspecified. We therefore do not know how many variables to introduce in a potentially equivalent expression of the form

**apply**(operator, *bvar*(var1,...,varn), *condition*(**apply**(in, **apply**(vector, var1, ..., varn), D)), **apply**(f, var1, ..., varn))<sup>3</sup>

It is, however, a perfectly legal MathML expression.

Secondly, and perhaps more importantly, the principles above carefully respect variable scopes – the most they do is split a variable scope into two parallel scopes. No additional scopes are ever introduced, nor are expressions moved across scope boundaries. By contrast, a rule that transforms

**apply**(operator, *bvar*(var1, ..., varn), *domainofapplication*(D), f)

to

**apply**(operator, *bvar*(var1, ..., varn), *condition*(**apply**(in, **apply**(vector, var1, ..., varn), D)), f)

<sup>3</sup> This *could* be expressed using an apply-to-list operator, as in

**apply**(operator, *bvar*(v), *condition*(**apply**(in, v, D)), **apply**(**apply-to-list**, f, v))

but no such concept is defined in MathML 2.

say, moves the entire expression  $D$  from outside the scope of the variables  $\text{var1} \dots \text{varn}$  to inside their scope. To make this legal, the rule would need to specify how to resolve naming conflicts between free variables in  $D$  and these bound variables. While OpenMath does specify that  $\alpha$ -conversion is allowed for bound variables in OpenMath expressions, thus making it possible, in principle, to specify such a rule, MathML never did go there, for good reasons.<sup>4</sup>

Beyond these fundamental flaws in a strict MathML that reduces domains of application to conditions rather than vice versa, the current draft of Strict MathML 3 violates a few more of the above principles that guided the development of MathML 2.

The proposal to rename **apply** to **bind** in case there is a *bvar* child violates Principle 2, the orthogonality and universal applicability of binding and ranging.

The proposal to allow only specific symbols as heads of expressions with a *bvar* child violates Principles 2 and 7, which state that the presence or absence of *bvars* should make no difference.

## 4.2 Comparison with OpenMath 2

OpenMath 2 has no first-class notion of a domain of application, although the MathML compatibility content dictionaries do contain constructors that emulate it. OpenMath 2 does have the equivalents of MathML 2's `apply` and of its `apply` with a *bvar* qualifier. The latter, however, severely restricts the type of heads it accepts (although it did not do so in OpenMath 1), while MathML 2 is completely open in this regard.

OpenMath 2 binding symbols in OpenMath Content Dictionaries tend to come in two varieties – the definite and the indefinite kind. Thus, where MathML only knows one symbol for integration, OpenMath 2 defines two symbols, one, **calculus1:defint**, for definite integration and one, **calculus1:int**, for indefinite integration. For the definite symbol, OpenMath defines two arguments, one for a function and one for the domain of application of that function. There is no consistency, however, to the order that these two types of arguments to definite operators appear in – sometimes (e.g., **arith1:sum**), the domain of application is the first argument, and sometimes (e.g., **calculus1:defint**) it is the second.

While indefinite symbols could be defined as binders that are allowed as heads of binding constructs, the structure of OpenMath mandates that the corresponding definite ones be defined as regular operators that take functional arguments. There are therefore only a very small number of binder symbols defined in OpenMath content dictionaries, whereas MathML includes a complete open-ended class of operators (“big” versions of  $n$ -ary functions) amongst those that take *bvar* qualifiers.

---

<sup>4</sup>  $\alpha$ -conversion is a semantic equivalence transformation, and should be defined as part of a formal semantics of a language like OpenMath or MathML. It should not be defined at the language's syntactic level, especially since variable names do frequently carry important pragmatic information (e.g., dimensionality) that should not be subjected to arbitrary removal.

OpenMath 2, as it is, thus has serious issues with the consistency Principle 7. Nevertheless, the OpenMath symbol for domain of application (**domainofapplication** in the **fns1** content dictionary) could allow for an easy integration between MathML and OpenMath. The question, however, is how and where to integrate it into an OpenMath expression. MathML's

**apply**(**example:int**, *bvar(x)*, *domainofapplication(D)*, f)

may correspond<sup>5</sup> in OpenMath 2 to

1. **omb**(**example:int**, *ombvar(x)*, **oma**(**fns1:domainofapplication**, f, D)),
2. **oma**(**fns1:domainofapplication**, **omb**(**example:int**, *ombvar(x)*, f), D),  
or
3. **omb**(**oma**(**fns1:domainofapplication**, **example:int**, D), *ombvar(x)*, f).

The first of these violates the scoping rules for the MathML expression, and should thus not be used. The second and third are possible, but the third violates the OpenMath 2 rule that forbids compound heads of binding expressions, leaving the second solution.

This is a problem, however. Intuitively, it is the first and third of these that come close to meanings of MathML's *domainofapplication*: it is either the function being integrated that is restricted to range over domain D, or it is the integration operator that is restricted to integrate only over domain D (the latter interpretation being quite common in functional analysis). It is emphatically not the *result* of the indefinite integration that is restricted to that domain. To the contrary: intuitively, if the domain of application is an open finite interval, the function that results from indefinite integration will almost certainly be applied to the outer limits of the open interval, which by definition are outside it.

The scoping rule violation of the first solution can be overcome by introducing an explicit lambda term, as the real OpenMath integration symbols require:

**oma**(**example:int**, **oma**(**domainofapplication**, **omb**(**fns1:lambda**, **ombvar**(x), f), D))

This correctly limits the domain of the integrand without violating scoping rules, however, integration is now indefinite rather than definite – the function that is integrated over is correctly limited to domain D, but nowhere does it now specify that definite integration over that entire domain is meant. The third solution thus presumably correctly captures the intuitive meaning of domain of application in this context – it is the domain over which the integrand is applied by the definite integration operator.<sup>6</sup>

<sup>5</sup> I'm using integration here for purely illustrative purposes. In reality, the *calculus1* OpenMath Content Dictionary defines both **int** and **defint** as operators that take the integrand as a functional argument, so that the above MathML expression would first need to be transformed according to Principle 2.

<sup>6</sup> Note that **fns1:domainofapplication** does not at this point have a precise definition; instead, it is simply stated that its intention is to mirror the *domainofapplication* construct of MathML. We are therefore free to try it on for a best fit.

It therefore appears that the domain of application can best be interpreted as modifying the head (operator) of the apply element that it appears in, specifying the domain over which that operator applies or reduces its functional argument.

## 5 Implications for MathML/OpenMath Version 3

The dichotomy between OpenMath and MathML-Content as two only superficially compatible standards for the semantic representation of mathematics on the Web has long been a nuisance, and it is important to fix it. However, it is just as important to do this right, and the above evidence suggests that current attempts at doing so may be severely, perhaps even fundamentally, flawed.

The basic ingredients for a successful unification appear to be in place. The idea of defining a core of MathML-Content to which any valid MathML-Content expression can be reduced while retaining semantic equivalence, and to adjust OpenMath to be fundamentally equivalent to such a core MathML-Content, deserves high praise. I hope that this paper can contribute to finding the right way to implement this idea.

It should have become clear from the above that, at least from the MathML perspective of such a merger, the concept of a domain of application must be retained as a first-class citizen of such a core MathML-Content. It is possible that there are good reasons for retaining a condition concept as well, but this may be optional. Variable binding, too, clearly needs to be retained, although it is not clear whether in the form of a **lambda** or in the form of a binding qualifier inside an **apply**. The above principles argue strongly against renaming **apply** in the context of binding, and they also imply that arbitrary heads need to be permitted in all circumstances.

Just because MathML does it, doesn't mean it's right, of course. I believe there are strong independent reasons to suspect that "domain of application" needs to be a first-class constructor in a content mathematics language: it is present in a wide range of mathematical concepts and it appears to be orthogonal semantically to the other two common first-class constructors (application and binding<sup>7</sup>). The compositionality principle implies in such a case that there should be a syntactic constructor in the language that encodes this concept. I have long argued [9] that adherence to the compositionality principle is a highly desirable trait for languages like MathML-Content and OpenMath.

For OpenMath, the challenge is thus to come up with a close equivalent of a "true" core MathML. The current drafts indicate that the OpenMath group has been attempting to muddle its way through this challenge, in part by co-opting MathML, but I hope that this paper will serve as an incentive for that group to "bite the bullet" and work for real compatibility with MathML. All of us who have worked on OpenMath need to recognize, I believe, that we made an honest mistake when we defined OpenMath 1 without a domain of application primitive

---

<sup>7</sup> Recall that the binding constructor was accepted as an unfortunate necessity for OpenMath only after years of intense discussions.

– and I include myself with those who made this mistake, having been present when this particular mistake was made.

Let us therefore admit that, in addition to **OMA** and **OMBIND** nodes, OpenMath needs, say, **OMDOA** nodes. As the above examples show, these need to be acceptable both in the heads and in the bodies of both **OMA** and **OMBIND** nodes. In particular, as the above example shows, this implies that heads of **OMBIND** nodes need to be allowed to contain compound objects – and I hope that the OpenMath community will refrain from making the same mistake again as in the move from OpenMath 1 to OpenMath 2 to put severe limits on the types of heads allowed for **OMBIND** nodes.

The absence of a first-class domain of application concept from OpenMath has led to a number of inconsistencies in OpenMath Content Dictionaries – adopting such a change, while perhaps a bit radical on the surface, should therefore significantly strengthen OpenMath, and do so not just by helping it become fully compatible with MathML.

## References

1. Buswell, S., Caprotti, O., Carlisle, D., Gaetano, M., Kohlhase, M. (eds.): The OpenMath Standard Version 2.0. <http://www.openmath.org/standard/om20-2004-06-30/>
2. Caprotti, O., Carlisle, D., Cohen, A. (eds.): The OpenMath Standard, Version 1.1b. <http://www.openmath.org/standard/om11/omstd11.pdf>
3. Carlisle, D., Ion, P., Miner, R. (eds.): Mathematical Markup Language (MathML) Version 3.0, W3C Working Draft 17 November 2008. <http://www.w3.org/TR/2008/WD-MathML3-20081117/>
4. Carlisle, D., Ion, P., Miner, R., Poppelier, N. (eds.): Mathematical Markup Language (MathML) Version 2.0 (2nd Edition), W3C Working Draft 11 April 2003. <http://www.w3.org/TR/2003/WD-MathML2-20030411/>
5. Devitt, S.: Re: errata and comments, chapters 2 and 4. [www-math@w3.org mailing list submission, 17 June 2003. http://lists.w3.org/Archives/Public/www-math/2003Jun/0042.html](http://lists.w3.org/Archives/Public/www-math/2003Jun/0042.html)
6. Strotmann, A.: Errata and comments, chapters 2 and 4. [www-math@w3.org mailing list submission, 8 May 2003. http://lists.w3.org/Archives/Public/www-math/2003May/0030.html](http://lists.w3.org/Archives/Public/www-math/2003May/0030.html)
7. Strotmann, A.: Bvar, condition, and domainofapplication. [www-math@w3.org mailing list submission, 13 May 2003. http://lists.w3.org/Archives/Public/www-math/2003May/0042.html](http://lists.w3.org/Archives/Public/www-math/2003May/0042.html)
8. Strotmann, A.: Errata and comments, chapter C (first installment). [www-math@w3.org mailing list submission, 8 May 2003. http://lists.w3.org/Archives/Public/www-math/2003May/0036.html](http://lists.w3.org/Archives/Public/www-math/2003May/0036.html)
9. Strotmann, A: Content Markup Language Design Principles. Doctoral dissertation, Dept. of Computer and Information Science, The Florida State University, Tallahassee, Florida (2003)
10. Strotmann, A: The categorial type of OpenMath objects. Mathematical Knowledge Management - Proceedings of the Third International Conference MKM 2004, September 19-21, 2004, Bialowieza, Poland