INFORMATION SOCIETY TECHNOLOGIES
(IST)
PROGRAMME

Project IST-2001-33562 MoWGLI

Report n. D4.a
MathML Rendering/Browsing Engine

Main Authors:
H. Naciri, L. Padovani

Project Acronym: MoWGLI
Project full title: Mathematics On the Web: Get it by Logic and Interfaces
Proposal/Contract no.: IST-2001-33562 MoWGLI

Contents

1 Overview

Interfaces for rendering and interacting with MathML markup [ABD+01, ABD+02] can be classified in the following categories:

General-purpose interfaces: an interface that adopts a common Web browser as the main component. These interfaces were briefly described in a different document[1] and will not be investigated further here;

Specialized interfaces: an interface that is targeted or otherwise optimized for more specific tasks which are not implementable (or whose implementation is not effective or does not meet the needed requirements) in common Web browsers.

This document describes two different engines for the creation of applications providing specialized interfaces for MathML. Each one of the two engines provides for an API that is peculiar to the development environment and graphical framework the engine is targeted to. Nonetheless, we can identify operations that are common in both cases and classify them in a more abstract way, as described in the next paragraphs.

Creation and initialization. The methods in this category are used to create *instances* of the engine. As initialization can be a complex operation involving large amounts of information (like font tables), it is usually performed once when the application is started or when the first instance is created, and may involve reading information from one or more configuration files.

Rendering. In the most general case the source data that is given to the rendering engine is a DOM document [HHN+00] representing the MathML document (or part of it) to be rendered. Starting from the DOM document the rendering engine creates a set of internal data structures that are suitable for rendering purposes, and finally displays the document in a *window*.

Retrieval. A *retrieval* operation is any operation that needs to retrieve the DOM element in the source MathML document corresponding to a bit of information pertaining to the graphical interface. Depending on whether the internal data structures of the rendering engine are exposed or hidden, this bit of information can range from a reference to a node in the internal data structure to a pair of coordinates in the window that displays the MathML document. Either way, the rendering engine is responsible for creating and maintaining *backward pointers* from its internal data structures to the corresponding DOM elements that generated them, so that the graphical interface represents just a *view* for the document, which is the real interesting information the interface should provide access for.

Note that the mechanism of backward pointers can be extended at will: in the previous paragraph we have described the mechanism between the rendering engine and the source MathML document. The MathML document, in turn, may have been generated in one or more steps from other sources (typically, other XML documents), hence the chain of backward pointers can extend accordingly. The strength and flexibility of a specialized interface allows the developer of the application to follow the chain of backward pointers up to the interesting source of information, be it the MathML presentation document, or some other XML resource.

[1] See the Preliminary Report on Application Scenarios and Requirement Analysis, Section 7.2.

Interaction. Rendering is a passive operation, in that the user does not play any active role. We define "interaction" as any operation other than the initial rendering of the document that involves an active role from the user side. We classify interactive operations in the following categories, in increasing order of complexity:

Clicking: the user clicks on the display area. The exact kind of action is determined by the application, typical examples being following an hyperlink, activating an `maction` element, opening a context-sensitive popup menu;

Selection: the user drags the mouse between two points of the display area to select a subtree of the source MathML document.

Editing: the user changes directly or indirectly the source document that the rendering engine is displaying.

The rendering engine will typically support these features in two ways:

1. generating one or more *asynchronous* signals (or events) corresponding to user actions, so that the application is notified on when and how the action occurs. The signal may carry information such as the coordinates of the mouse pointer, or directly a reference to the DOM element under the pointer;

2. providing for methods that the application can invoke *synchronously* for achieving effects related to interaction or for querying the rendering engine about information needed to interaction.

Editing is not a main concern in the context of MoWGLI, hence we will not discuss any special feature or requirement in detail.

Conversely, selection is a crucial feature especially because any complex form of selection is typically very hard to implement and customize using a browser-oriented interface. In particular we refer to *semantic selection*, that is the selection of a MathML presentation subtree that corresponds to a *significant* subtree from which the MathML presentation was automatically created. The exact definition of "significant" is of course application and context dependent. For example, in the simple case of a transformation from MathML content to MathML presentation, it is easy to realize that there need not be a one-to-one correspondence between presentation and content subtrees. Assuming that the presentation markup also carries backward pointers to the corresponding content markup (but not every MathML presentation element will have a backward pointer), semantic selection can be performed by examining the element which the pointer is currently on, and then by going up the DOM tree until an element that has the backward pointer is eventually found.

2 GtkMathView

GTK+ [GTK] is an object-oriented framework developed in C for the creation of Graphical User Interfaces. The basic components of the framework, the so-called *widgets*, implement objects like buttons, menus, labels, text editing areas, and so on. GtkMathView [Pad02] is a GTK+ widget for rendering and interacting with MathML markup (see Figure 1).

Although GtkMathView is, by its own definition, related to the GTK+ framework (we eventually had to commit to a development platform that was convenient for our purposes),

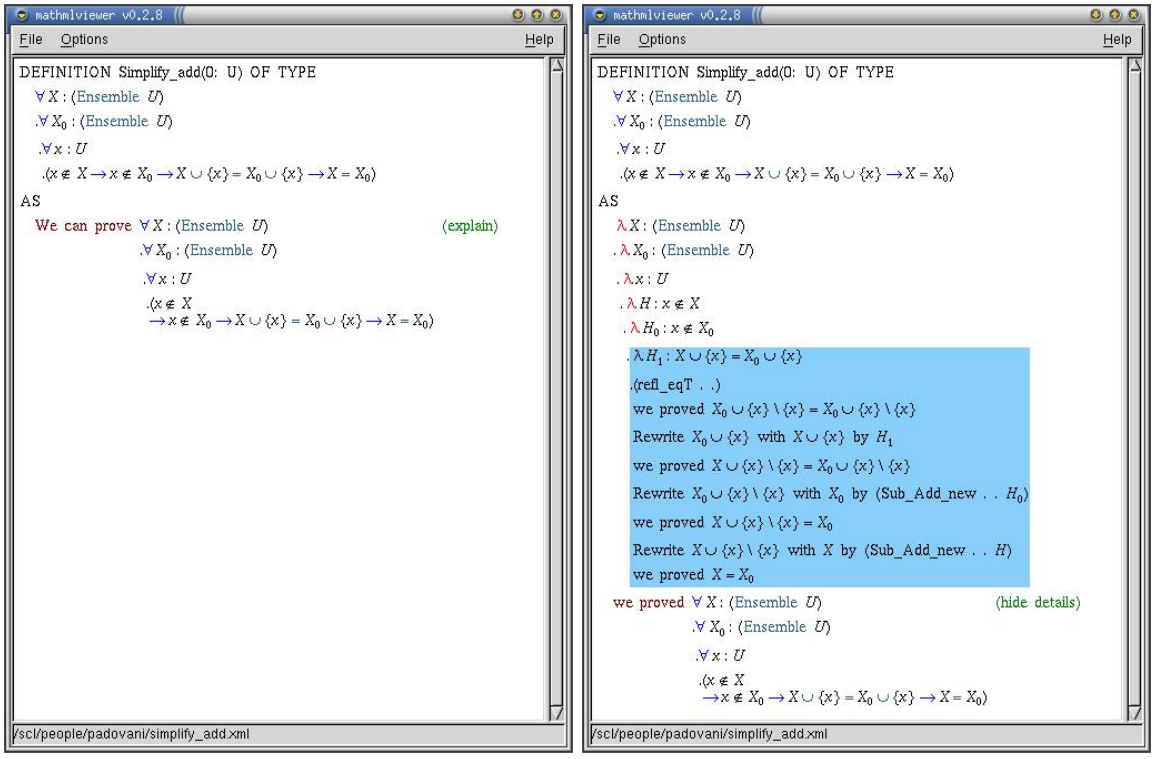


Figure 1: A theorem of the Coq library rendered by GtkMathView. On the left hand side the theorem is in its contracted form, only the statement is shown. It is possible to expand the proof by clicking on the green "explain" token, thus revealing more details (on the right hand side). It is also shown the selection mechanism at work (on the right hand side, the light-blue shaded part).

its internal architecture is designed as to isolate as much as possible platform dependencies. A large part of GtkMathView, the one dealing with proper formula layout, is completely platform independent. On top of this internal, hidden library of C++ classes are two different interfaces, one for GTK+ and another one that allows to render MathML documents to PostScript. The implementation of a new interface is a relatively easy and straightforward task.

2.1 GtkMathView Installation

GtkMathView is available in the following forms:

- as precompiled library files for the Debian[2] distribution of Linux. This is the easiest way to install GtkMathView as the Debian installation system will also install any libraries that GtkMathView depends on. There are also unofficial packages for the RedHat[3] Linux distribution, but these can be out-of-date as we are not directly responsible for them;

[2] `http://www.debian.org`
[3] `http://www.redhat.com`

- as source code from GtkMathView's home page.[4] Only stable releases can be found in this site;
- as source code from the GNOME CVS repository, which can be accesses anonymously via CVS at the following address

  ```
  :pserver:anonymous@anoncvs.gnome.org:/cvs/gnome
  ```

  The module name is `gtkmathview`. The CVS repository can also be browsed online at the address `http://cvs.gnome.org/lxr/source/gtkmathview/`.

In order to compile and use GtkMathView, the following libraries must be installed:

- GTK+ (version 1.x).[5]
- libxml2 (at least version 2.4.x).[6] This library provides a C API for parsing XML documents and accessing their structure and content;
- Gdome2 (at least version 0.7.3).[7] This library builds on top of libxml2 and provides a DOM-compliant C API for accessing and modifying the structure and the content of XML documents;
- GMetaDOM (at least version 0.5.1).[8] This library provides C++ and Ocaml bindings for Gdome2;
- t1lib (version 1.3.1).[9] This library is optional but recommended; it provides support for Type1 fonts and exportation of MathML documents into PostScript.

2.2 Configuration

The widget can be partly configured after instantiation by using the provided methods described in Section 2.3. However, most of the configuration is specified in an external file whose path can be set in the environment variable `MATHENGINECONF`. In case the environment variable is not set, the widget will try to read the configuration file from the default location, which typically is `/usr/local/share/gtkmathview`.

2.2.1 Main configuration file

The main configuration file is an XML document whose root element must be `math-engine-configuration`. The children of the root element determine the value of a number of parameter that are described below:

`dictionary-path`: the content of this element is the full path of the XML file describing the content of the operator dictionary used by GtkMathView.

[4] `http://helm.cs.unibo.it/mml-widget/`
[5] `http://www.gtk.org`
[6] `http://xmlsoft.org`
[7] `http://gdome2.cs.unibo.it`
[8] `http://gmetadom.sourceforge.net`
[9] `ftp://sunsite.unc.edu/pub/Linux/libs/graphics/`

There can be more instances of the `dictionary-path` element, each pointing to a different dictionary. If an entry of the dictionary is defined multiple times, possibly within different dictionaries, the last definition is the one that matters. This way it is possible to define the suggested operator dictionary as by the MathML specification, and then customize it depending on the application needs.

The format of the operator dictionary will be detailed in Section 2.2.2.

`font-configuration-path:` the content of this element is the full path of the XML file describing the font configuration.

There are usually several instances of the `font-configuration-path` element.

The format of the font configuration file will be detailed in Section 2.2.3.

`t1-config-path:` the content of this element is the full path of the `t1lib` font configuration file. The format of this file is detailed in the `t1lib` documentation and will not be described here. Also, this entry is only necessary if the use of Type 1 fonts is required.

`font-size:` the attribute `size` of this element can be used to set the default value of font size used for rendering the MathML document. The number must be followed by the appropriate unit (typically points). For example, to set the default font size to `12` points the `size` attribute must be set to the string "`12pt`". The set of valid units is the same as the set of units allowed in MathML documents; it is described in the MathML specification. An application using GtkMathView can override the default font size by invoking the `_set_font_size` method which is described in Section 2.3.

`color:` default color of the rendered document. The attributes `foreground` and `background` determine the foreground and background colors respectively. The format of the two attributes is the same as for color specifications on MathML elements accepting the `mathcolor` attribute. In particular, it is possible to use one of the named colors defined by the HTML specification, or to use an RGB format (a '#' sign followed by 3 or 6 hexadecimal digits).

`link-color:` default color of tokens with an `xlink:href` attribute. The format of the two `foreground` and `background` attributes is the same as for the `color` element.

`select-color:` color combination used for selections. The format of the two `foreground` and `background` attributes is the same as for the `color` element.

2.2.2 Operator Dictionary

The operator dictionary defines the default value of a number of attributes that applies to `mo` elements. The exact definition of the operator dictionary along with the attributes that can be set is described in the MathML specification.[10]

The dictionary XML file is made of a `dictionary` element with an arbitrary number of `operator` children. For each operator the following attributes can be used: `name` (this is required), `form` (this is also required and can be one of `prefix`, `infix`, or `postfix`), `fence`, `separator`, `lspace`, `rspace`, `stretchy`, `symmetric`, `maxsize`, `minsize`, `largeop`,

[10] See `http://www.w3.org/TR/MathML2`

`movablelimits`, `accent`. The syntax for the attributes is the same as the one specified in the MathML recommendation.

In addition, two extension attributes `tspace` and `bspace` can be defined. They represent the amount of space respectively at the top and at the bottom of an operator. They are usually ignored, unless the widget has been configured with the `--enable-extensions` flag set.

A sample entry for the operator dictionary is the following, describing the default attributes for an open parenthesis when it is in prefix position (that is, the parenthesis is the first child of a (possibly inferred) `mrow` element):

```
<operator name="(" form="prefix" fence="true" stretchy="true"
                                 lspace="0em" rspace="0em"/>
```

2.2.3 Font Configuration

Font configuration defines the mapping from Unicode[11] characters to glyph indexes inside system fonts. Although font configuration should be done once and for all, and the widget already comes with a sensible configuration for most math fonts, we will describe the font configuration format so that users know how to extend their configuration, or to correct the existing one in case of errors.

The font configuration XML file is made of a `font-configuration` element with an arbitrary number of `font` and `map` children. `font` children define how abstract font descriptions are mapped into system font descriptions, whereas `map` children define an actual mapping of Unicode characters into glyph indexes for a particular font. Every `font` abstract description *must have* a reference to a `map` element. The reason why font abstract descriptions and maps are kept separate is that many different font abstract descriptions may share the same font map, hence it is convenient to use an indirection instead of replicating the map each time it is needed.

Each `font` element has the following attributes attributes:

Font classification:

`type`: type of the font. Currently `x` (for X) or `ps` (for PostScript, Type1) fonts are supported;

`mode`: mode for which the font should be used. May be one of `any`, `text`, or `math`.

`mapping`: reference to the corresponding `map` element. It can be an arbitrary non-empty string.

Font abstract description:

`family`: the font family. This field is not fixed, but the most common values are `serif`, `sans-serif`, `monospace`;

`style`: either `normal` or `italic`;

`weight`: either `normal` or `bold`;

`size`: preferred size at which the font should be used. If not specified the font will be magnified to the desired size.

[11] See `http://www.unicode.org`

Specific font data. The set of valid attributes depends on the `type` attribute. For X fonts, the attributes are `x-foundry`, `x-family`, `x-weight`, `x-slant`, `x-width`, `x-style`, `x-pixels`, `x-points`, `x-hres`, `x-vres`, `x-spacing`, `x-avgwidth`, `x-registry`, `x-encoding`. Their syntax follows the syntax for X font specifications.

For PostScript (or Type1) fonts, the attributes are `ps-name` and `ps-file` specifying the PostScript name of the font and the file name that contains the font, respectively.

Each `map` element has an `id` attribute that font abstract specifications can refer to and has an arbitrary list of `single`, `range`, `multi`, or `stretchy` children. Whenever Unicode characters or glyph indexes need to be referenced from the maps, they can be given in hexadecimal (with `0x` prefix), octal (with `0` prefix), or decimal (in all the other cases). Unicode characters can also be specified by simply inserting the character (or a character reference) into the attribute.

A `single` child specifies the mapping for a single Unicode character. The Unicode character is specified in the `code` attribute, its index in the `index` attribute.

A `range` child specifies the mapping for a range of Unicode characters. The first and last characters (inclusive) in the range are specified in the `first` and `last` attributes, respectively. The attribute `offset` specifies the index of the glyph corresponding to the `first` character.

A `multi` child specifies the mapping for a range of Unicode characters whose glyph indexes do not follow the same order, or are not all available. In this case the `index` attribute is a space separated list of glyph indexes, each related to the corresponding Unicode character in the range. If there is no glyph corresponding to a given Unicode character, the constant `-1` must be used.

A `stretchy` child specifies the mapping of a stretchable Unicode character, that is a character that can be formatted in different sizes, depending on the context. The `code` attribute specifies the Unicode character, and the `direction` attribute specifies whether the character can be stretched horizontally (`horizontal`), vertically (`vertical`), or in both directions (`both`). A stretchy character has a mandatory `simple` child optionally followed by a `compound` child. The `simple` child has an `index` attribute that specifies the glyph indexes (separated by spaces) corresponding to the character as a whole, in increasing order of size. The `compound` child has an `index` attribute that specifies the glyph indexes (separated by spaces) corresponding to the dismantled pieces that can be combined to build up the character. There must be 4 entries corresponding to the leftmost (or bottommost) piece, the middle piece, the rightmost (or topmost) piece, and the piece that can be repeated at will to make the symbol as hight (or wide) as required.

For example, a specification like

```
<stretchy code="(" direction="vertical">
  <simple index="0000 0020 0022 0040"/>
  <compound index="0060 -1 0100 0102"/>
</stretchy>
```

says that the open round parenthesis is available as simple glyphs at the positions `0000`, `0020`, `0022`, `0040`. Those glyphs are listed in increasing order of size. In case there is the need of an even bigger parenthesis, it can be combined by using the `0060` glyph for the bottommost piece, `0100` glyph for the topmost piece, and `0102` as the glyph that can be repeated at will.

2.3 GTK+ Interface

The GTK+ interface provides a set of *high-level* methods for the creation, the usage, and the destruction of instances of the GtkMathView widget.

2.3.1 Macros

`GTK_TYPE_MATH_VIEW`
Retrieves the GTK+ identifier corresponding to the GtkMathView class. It is not normally needed, unless one is extending the GtkMathView class.

`GTK_MATH_VIEW`(object)
Performs a safe cast from a `GtkWidget* object` to a `GtkMathView*` object.

`GTK_MATH_VIEW_CLASS`(klass)
Casts a GTK+ class `klass` to the GtkMathView internal class object. This macro is used internally, and is not normally needed by the user of the library.

`GTK_IS_MATH_VIEW`(object)
Returns `TRUE` if `object` is an instance of the `GtkMathView` class, `FALSE` otherwise. It is normally used as a runtime check before casting `object` to the `GtkMathView` type.

2.3.2 Methods

```
GtkType
gtk_math_view_get_type()
```
Allocates a new GTK+ type for the `GtkMathView` class. This function is used internally and is not normally needed by the user of the library.

```
GtkMathView*
gtk_math_view_new(GtkAdjustment* h_adj, GtkAdjustment* v_adj)
```
Creates a new instance of the `GtkMathView` class. The two parameters passed to the constructor are used to associate scrolling bars with the drawing area of the widget. In case the parameters are `NULL`, the widget will allocate its own scrolling bars.

```
void
gtk_math_view_freeze(GtkMathView* widget)
```
After the invocation of this method any update of the widget is postponed until the invocation of the `_thaw` method. It can be useful to stop automatic update of the widget in those cases in which a large number of modifications to the source document tree are done, so that it would be inefficient and useless to update the view at every single modification.

```
void
gtk_math_view_thaw(GtkMathView* widget)
```
Updates the widget after the invocation of the `_freeze` method. Any modification to the source document tree that occurred while the widget was "frozen" is displayed at this time.

```
gboolean
gtk_math_view_load_uri(GtkMathView* widget, const gchar* uri)
```

Loads the MathML document identified by `uri` and displays it in the view.

`gboolean`
`gtk_math_view_load_doc(GtkMathView* widget, GdomeDocument* doc)`
Displays the MathML document `doc` in the widget's window.

`gboolean`
`gtk_math_view_load_tree(GtkMathView* widget, GdomeElement* elem)`
Displays the MathML fragment whose root is `elem` in the widget's window. `elem` must be a `math` element in the MathML namespace.

`void`
`gtk_math_view_unload(GtkMathView* widget)`
Releases the widget's resources relative to the MathML document that is currently displayed. The widget's window is erased.

`void`
`gtk_math_view_load_select(GtkMathView* widget, GdomeElement* elem)`
Marks the element `elem` as "selected." This has the effect of highlighting the portion of the widget's window that include the element itself, along with all its children.

`void`
`gtk_math_view_load_unselect(GtkMathView* widget, GdomeElement* elem)`
Marks the element `elem` as "not selected."

`gboolean`
`gtk_math_view_load_is_selected(GtkMathView* widget, GdomeElement* elem)`
Queries about the selection status of the element `elem`, returns `TRUE` if `elem` is selected, `FALSE` otherwise.

`GdomeElement*`
`gtk_math_view_get_element_at(GtkMathView* widget, gint x, gint y)`
Returns a reference to the innermost element whose extents comprise the point at coordinates (`x`, `y`), or `NULL` if no such element exists.

`gboolean`
`gtk_math_view_get_element_coords(GtkMathView* widget, GdomeElement* elem,`
`gint* x, gint* y)`
Returns the coordinates of the upper-left corner of the rectangular area covered by the element `elem`. If `elem` does not belong to the rendered document, or if it has no corresponding visual object, the method returns `FALSE` and `x` and `y` are left unchanged.

`gboolean`
`gtk_math_view_get_element_rectangle(GtkMathView* widget, GdomeElement* elem,`
`GdkRectangle* rect)`
Returns the position and the extents of the rectangular area covered by the element `elem`. If `elem` does not belong to the rendered document, or if it has no corresponding visual object, the method returns `FALSE` and `rect` is left unchanged.

`gint`
`gtk_math_view_get_width(GtkMathView* widget)`
Returns the horizontal extent of the viewing area.

`gint`
`gtk_math_view_get_height(GtkMathView* widget)`
Returns the vertical extent of the viewing area.

`void`
`gtk_math_view_get_top(GtkMathView* widget, gint* x, gint* y)`
Returns the coordinates of the upper-left corner of the viewing area. When the MathML document is displayed for the first time, the coordinates are set to $(0, 0)$. As the document view is scrolled horizontally or vertically, the coordinates of the upper-left corner change accordingly.

`void`
`gtk_math_view_set_top(GtkMathView* widget, gint x, gint y)`
Sets the coordinates of the upper-right corner of the viewing area to $(\mathtt{x}, \mathtt{y})$.

`void`
`gtk_math_view_set_adjustments(GtkMathView* widget, GtkAdjustment* h_adj,`
`GtkAdjustment* v_adj)`
Sets the horizontal and vertical adjustments of the viewing widget to `h_adj` and `v_adj` respectively. The adjustments are particular GTK+ objects storing the state of the scrolling bars. This method is not normally used by the client application.

`void`
`gtk_math_view_get_adjustments(GtkMathView* widget, GtkAdjustment** h_adj,`
`GtkAdjustment** v_adj)`
Retrieves the horizontal and vertical adjustments of the viewing widget. This method is not normally used by the client application.

`GdkPixmap*`
`gtk_math_view_get_buffer(GtkMathView* widget)`
Returns the internal GTK+ pixmap that the widget uses for double buffering.

`GdkFrame*`
`gtk_math_view_get_frame(GtkMathView* widget)`
Returns the internal GTK+ widget used to render the frame around the viewing area.

`GtkDrawingArea*`
`gtk_math_view_get_drawing_area(GtkMathView* widget)`
Returns the GTK+ widget representing the drawing area that the widget uses for rendering on the screen. The returned widget can be used for setting signal handlers other than those generated directly by GtkMathView.

`void`
`gtk_math_view_set_font_size(GtkMathView* widget, guint size)`
Sets the default font size to `size`. The font size is specified in *points*.

`guint`
`gtk_math_view_get_font_size(GtkMathView* widget)`
Retrieves the default font size.

`void`
`gtk_math_view_set_anti_aliasing(GtkMathView* widget, gboolean flag)`
Enables or disables anti-aliasing according to the value of `flag`. Anti-aliasing of characters is only available when Type 1 fonts are used.

`gboolean`
`gtk_math_view_get_anti_aliasing(GtkMathView* widget)`
Returns the status of the anti-aliasing flag.

`void`
`gtk_math_view_set_transparency(GtkMathView* widget, gboolean flag)`
Enables or disables transparency according to the value of `flag`. When transparency is enabled, the widget renders the glyph without rendering the background. This way, if two or more glyphs overlap for any reason, the glyphs are rendered correctly. However, transparency is supported only by Type 1 fonts and it is usually slower than opaque rendering.

`gboolean`
`gtk_math_view_get_transparency(GtkMathView* widget)`
Returns the status of the transparency flag.

`void`
`gtk_math_view_set_log_verbosity(GtkMathView* widget, gint level)`
Sets the level of verbosity of the widget. The higher the level, the more messages are displayed on the console. There are currently 4 levels of verbosity represented by the integer numbers from 0 to 3. They stand for `ERROR`, `WARNING`, `INFO`, and `DEBUG`. For example, when the verbosity level is set to 0, only serious errors are reported. When the verbosity level is set to 2, errors, warnings and informative messages are displayed, but debugging messages are not.

`gint`
`gtk_math_view_get_log_verbosity(GtkMathView* widget)`
Returns the current level of verbosity.

`void`
`gtk_math_view_export_to_postscript(GtkMathView* widget, gint width, gint height,`
`gint h_margin, gint v_margin, gboolean no_colors, FILE* output)`
Creates a PostScript file with the rendered MathML document. The arguments `width` and `height` determine the horizontal and vertical extent (in pixels) of the virtual drawing area that is available for rendering. The `h_margin` and `v_margin` determine the margin (in pixels) of the virtual drawing area that is available for rendering. If `no_colors` is set to `TRUE` the document is rendered in black and white and any information about colors is discarded. `output` is the file on which the PostScript document is written.

`void`
`gtk_math_view_set_font_manager_type(GtkMathView* widget, FontManagerId id)`

Sets the type of font manager used by the widget. There are currently two different choices, `FONT_MANAGER_GTK` and `FONT_MANAGER_T1`. When the `FONT_MANAGER_GTK` is set, the widget uses the fonts that are available by means of the X server (or the X font server). When the `FONT_MANAGER_T1` is set, the widget uses the fonts that are available by means of the `t1lib` library, which handles Type 1 fonts (sometimes referred to as PostScript fonts).

`FontManagerId`
`gtk_math_view_get_font_manager_type(GtkMathView* widget)`
Returns the current font manager type.

2.3.3 Signals

`"click"(GtkWidget* widget, GdomeElement* elem, int state)`
Emitted whenever the user click on the displaying window. `elem` is the reference to the DOM element on which the used clicked, or `NULL` if the area clicked does not correspond to any MathML element.

`"select_begin"(GtkWidget* widget, GdomeElement* elem, int state)`
Emitted when the user starts a selection, that is the first mouse button is pressed and the mouse is moved a bit from the original location. The `elem` and `state` arguments have the same meaning as for the `"click"` signal.

`"select_over"(GtkWidget* widget, GdomeElement* elem, int state)`
Emitted when the user moves the mouse while keeping the first button pressed. `elem` is the element which the mouse pointer is currently on.

`"select_end"(GtkWidget* widget, GdomeElement* elem, int state)`
Emitted when the user releases the first mouse button after a selection operation.

`"select_abort"(GtkWidget* widget)`
Emitted if the user presses any mouse button other than the first one while selection is in progress. This signal terminates the selection process, no other selection-related event will be emitted until the first mouse button is released and pressed again.

`"element_over"(GtkWidget* widget, GdomeElement* elem, int state)`
Emitted as the mouse pointer is moved on the displayed document, no matter of the state of the mouse buttons.

2.4 Interactivity Support

GtkMathView's interactivity support can be summarized in three aspects, *selection*, *point-and-click*, and *editing*, which we will define more precisely in the sections that follows.

Interactivity support often implies re-rendering of the displayed MathML document. As this can be an expensive operation, especially when several actions take place in rapid succession, GtkMathView provides for two methods (`_freeze` and `_thaw`) that delay any re-computation and update of the displaying window until the application has terminated the action.

As a matter of fact, due to the current design of GtkMathView's architecture, the use of these two methods is *mandatory* in that the displayed window is guaranteed to be updated

correctly only if the two methods are properly used. Although we perceive this as a limitation, the points in favor are the following:

1. we found that in most cases the handling of the action already justifies the use of `_freeze` and `_thaw`, as a way of improving performances;
2. they are lightweight methods that have a negligible impact in the user application;
3. because of the DOM-driven architecture, the implementation of the alternative mechanism seems to violate the clear separation of the platform-independent engine from the graphical interface.

It is safe to nest calls to `_freeze` and `_thaw` methods at any level.

2.4.1 Selection

By *selection* we mean the possibility for the user to distinguish one or more DOM elements from the others in a MathML document. The typical visual feedback for selected elements is that of displaying them with a different background color.

Selection support in GtkMathView consists of three methods and four signals. The methods are needed to set and query about the selection status of a particular MathML element, as this information is hidden within the internal data structures of GtkMathView and is not available as part of the MathML document itself. Selection is a boolean property: an element can be either selected or not, there are no multiple levels of selection.

Both `_select` and `_unselect` operate recursively on the given MathML element and also on all its descendants, but the `unselect` method can be used to perforate a previously selected element thus leaving a "hole" within a selection. This mechanism can be exploited for representing *patterns* of elements within the MathML documents.

The signals related to selection are `select_begin`, `select_end`, `select_over`, and `select_abort`. The first three signals have two arguments, the MathML element on which the signal has been emitted and the status of control keys on the keyboard. The four signals are fired in disjoint sequences matching the following regular expression:

$$\texttt{select\_begin}\ (\texttt{select\_over})^*\ (\texttt{select\_end} \mid \texttt{select\_abort})$$

which is to be read as follows: selection begins when the user presses the first mouse button and moves it a bit from the original position. As the user moves the mouse with the button pressed, an arbitrary number of `select_over` signals is emitted. Selection can terminate in two cases: either the user releases the button (`select_end`) or he/she presses another mouse button, hence aborting the selection (`select_abort`).

An interface for which multiple selections are required can check the status of control keys on the keyboard in order to determine whether a new selection sequence replaces a previous selection or adds a new selection to it.

Support for semantic selection. The widget does not highlight automatically the parts of the document on which the user is dragging the mouse. It is complete responsibility of the application to handle the selection signals and to invoke the `_select` method appropriately. Although this puts some burden on the application side, it also enables the maximum flexibility, as selection may be constrained in ways that, in the most general case, are infeasible to hard code within GtkMathView.

2.4.2 Point-and-click Functionalities

Point-and-click is supported by the `click` signal, which is emitted when the users clicks on the MathML document. The two arguments of the signal are the deepest MathML element on which the mouse was placed at the time of the click or `NULL` if there is no such element, and the status of the control keys on the keyboard.

Among the possible usages for this signal are the activation of hyperlinks and the management of `maction` elements. When the user clicks on the document, the event handler can look for a MathML element that has a `xlink:href` attribute set, and render that document. The search is typically done by starting from the element provided by the signal, and possibly climbing up the chain of elements until one with the required attribute is found, or the root element is reached, or any other application-specific condition applies.

Activation of `maction` elements works in the same way. Currently only the `toggle` action is supported by GtkMathView. In particular, once the `maction` element is found, it is possible to increase the number found in the `selection` attribute, which determines which of the `maction`'s children is displayed. If the `selection` is increased beyond the actual number of children, GtkMathView will recast it into a valid range via a modulo operation. This way the application code for handling `maction` does not have to know the exact number of children, and is thus simplified. Note however that this behavior is not mandated by the MathML recommendation, and can differ in other MathML rendering engines. Once the `selection` attribute is set with the new value, GtkMathView will recompute automatically the document's layout.

Possible conflicts in case the `xlink:href` is set on an `maction` element must be resolved by the signal handler.

2.4.3 Editing

The notion of *editing* in the context of GtkMathView is very limited. It simply refers to the fact that GtkMathView reacts automatically as the source MathML document changes, but GtkMathView itself does not enforce any constraints on how and when the document can change. The management of `maction` can be seen as a very particular kind of editing, as it involves a modification of the source document tree.

GtkMathView implements a number of internal mechanisms that try to optimize rendering, in the sense of minimizing the amount of computation that is needed to re-render a document after a modification has occurred.

Note also that in some cases local modifications may have non-local effects. For instance, modifying the content of a table cell may cause the re-computation of the whole table layout, as MathML attributes for table can specify constraints among cells in different columns or rows.

2.5 Sample Application

In this section we show a minimal GTK+ application that uses GtkMathView for displaying a MathML document. The application is intentionally small and only shows a small subset of GtkMathView capabilities. More complex examples, including sophisticated selections and `maction` handling, can be found in the GtkMathView source distribution.

2.5.1 The Source Code

The application begins, as usual, with some `#include` directives that load the necessary `.h` files. Note in particular that the `gtkmathview.h` header file must be included in order to use the widget.

```
#include <stdio.h>
#include <gtk/gtk.h>
#include <gtkmathview.h>
```

Next we declare a number of global variables that represent the main components of the graphical interface. The reason why these variables are global is that we may need to reference them from outside the `main` function, where they are initialized.

```
static GtkWidget* window = NULL;
static GtkWidget* main_vbox = NULL;
static GtkWidget* main_area = NULL;
static GtkWidget* scrolled_area = NULL;
```

We also need two global variables to support selection:

```
static GdomeElement* first_selected = NULL;
static GdomeElement* root_selected = NULL;
```

The following is an auxiliary function that loads the MathML document whose URI is specified in `name` inside the widget.

```
static int
load_document(const char* name)
{
  GtkMathView* math_view = GTK_MATH_VIEW(main_area);

  g_return_val_if_fail(name != NULL, -1);
  g_return_val_if_fail(math_view != NULL, -1);

  return gtk_math_view_load_uri(math_view, name);
}
```

The `click` function will be responsible for handling `click` signals as they are emitted. Note that the function takes three arguments: the GtkMathView widget that has generated the signal (`math_view`), the MathML element on which the user has clicked (`elem`), and finally the status of control keys.

```
static void
click(GtkMathView* math_view, GdomeElement* elem, gint state)
{
  g_return_if_fail(math_view != NULL);

  printf("*** click signal: %p %x\n", elem, state);
```

```
  if (elem != NULL)
    {
      /* do something with the element */
    }
}
```

The implementation of the selection mechanism is slightly more complicated as it involves handling at least two different signals. Upon receiving the `select_begin` signal the current selection, if present, is deleted, and the two global variables `first_selected` and `root_selected` are initialized with the element `elem` under the mouse pointer at the time selection was initiated. The element, if not `NULL`, is highlighted.

Note that the whole body of the signal handler begins with a `_freeze` operation and ends with a `_thaw` operation. This means that the display window is not updated until the signal handler terminates, with two advantages: the method is more efficient as GtkMathView updates only once, after two operations have been completed, and also we avoid the problem of flickering in case the previously selected area overlaps with the new one.

```
static void
select_begin(GtkMathView* math_view, GdomeElement* elem, gint state)
{
  gtk_math_view_freeze(math_view);

  if (root_selected != NULL)
    gtk_math_view_unselect(math_view, root_selected);

  first_selected = root_selected = elem;

  if (root_selected != NULL)
    gtk_math_view_select(math_view, elem);

  gtk_math_view_thaw(math_view);
}
```

Next we have to handle the signal emitted as the user drags the mouse keeping the button pressed. The idea is that `first_selected` remembers the element on which selection was initiated. As the mouse pointer is moved over the document, it crosses other elements. From `first_selected` and the current element `elem` it is possible to compute the smallest DOM subtree selected. In the sample code this operation is accomplished by the `common_ancestor`, which is left unspecified here.

```
static void
select_over(GtkMathView* math_view, GdomeElement* elem, gint state)
{
  gtk_math_view_freeze(math_view);

  if (root_selected != NULL)
    gtk_math_view_unselect(math_view, root_selected);
```

```
  root_selected = common_ancestor(first_selected, elem);

  if (root_selected != NULL)
    gtk_math_view_select(math_view, elem);

  gtk_math_view_thaw(math_view);
}
```

Note that the sample code does not implement any mechanism for semantic selection, but the idea is that the two signal handlers (as well as the remaining ones not shown here for simplicity), can be made as complex as the application needs. In particular, the computation of the selected root (`common_ancestor`) can be specialized so to search the smallest DOM subtree that meets the desired requirements as the application needs.

In order to complete the application, let us examine the `main` function, which creates the graphical interface, along with an instance of the GtkMathView widget, attaches the `delete_event` to the closing button of the window so that the window is automatically destroyed when the user closes it, and connects the signal handlers. Then the document whose URI is passed on the command line of the application is loaded into GtkMathView, and the control is finally transferred to GTK+.

```
main(int argc, char *argv[])
{
  gtk_init(&argc, &argv);

  window = gtk_window_new(GTK_WINDOW_TOPLEVEL);
  gtk_signal_connect(GTK_OBJECT(window), "delete_event",
                     (GtkSignalFunc) gtk_main_quit, NULL);
  gtk_widget_show(window);

  main_vbox = gtk_vbox_new(FALSE, 1);
  gtk_container_border_width(GTK_CONTAINER(main_vbox), 1);
  gtk_container_add(GTK_CONTAINER(window), main_vbox);
  gtk_widget_show(main_vbox);

  main_area = gtk_math_view_new(NULL, NULL);
  gtk_widget_show(main_area);

  gtk_signal_connect_object (GTK_OBJECT (main_area),
                             "click", GTK_SIGNAL_FUNC(click),
                             (gpointer) main_area);

  gtk_signal_connect_object (GTK_OBJECT (main_area),
                             "select_begin", GTK_SIGNAL_FUNC(select_begin),
                             (gpointer) main_area);

  gtk_signal_connect_object (GTK_OBJECT (main_area),
                             "select_over", GTK_SIGNAL_FUNC(select_over),
```

```
                                (gpointer) main_area);

  scrolled_area = gtk_scrolled_window_new(NULL, NULL);
  gtk_scrolled_window_set_policy(GTK_SCROLLED_WINDOW(scrolled_area),
                                 GTK_POLICY_AUTOMATIC, GTK_POLICY_ALWAYS);
  gtk_widget_show(scrolled_area);
  gtk_container_add(GTK_CONTAINER(scrolled_area), main_area);
  gtk_box_pack_start(GTK_BOX(main_vbox), scrolled_area, TRUE, TRUE, 0);

  load_document(argv[1]);

  gtk_main();
}
```

A note on memory management. It is important to mention that the objects of the DOM implementation (GDome2 [CP02a, CP02b]) used in the sample code are subject to a memory management system that is based on reference counting. This means that every time a pointer to a DOM object (like a `GdomeElement*` in the example) is stored in some variable, the application has to explicitly increment its reference counter so that the object is not deallocated. For the sake of brevity every operation involving the reference counting mechanism has been omitted, but it is crucial to use it correctly in every application, no matter how simple the operations are. In this respect, the use of the corresponding C++ library (GMetaDOM) simplifies significantly the work of the programmer, as the DOM objects are accessed by means of *smart pointers* that take care of the reference counter automatically, without explicit intervention from the programmer's side.

2.5.2 Compilation and Execution

The easiest way to compile the source code of the example in Section 2.5 is to use the script `gtkmathview-config` provided with GtkMathView, which provides automatically the compilation and linking flags to be passed to the C compiler.

Assuming the application is inside a file called `main.c`, and that the compiler used is GCC[12], compilation and linking are done with

```
gcc ‘gtkmathview-config --cflags --libs‘ main.c
```

It is now possible to execute the application and view a MathML document:

```
./a.out <mathml-document.xml>
```

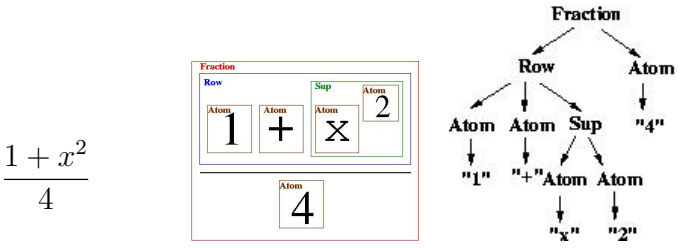
3 FIGUE and MathML rendering

3.1 What is FIGUE?

FIGUE is an incremental interactive 2-dimensional extensible JAVA library developed in INRIA. FIGUE gives a two dimensional display of structured objects. Like the majority of layout tools, FIGUE is based on the idea of layout boxes [Knu90] to represent structured objects

[12] See `http://gcc.gnu.org`

of the displayed document. FIGUE directly manipulates a box-tree which describes how to layout the objects on a page and memorizes the dependence relations (hierarchy of inclusion) between graphical objects. Each node in the tree represents a box (associated with a graphical constructor) including all its child boxes, and the leaves are tokens (basic units like strings) appearing in the final displayed document. Figure 2 shows a simple example of a box representation for a mathematical formula.



(a) (b) (c)

Figure 2: (a) Mathematical formula. (b) Corresponding boxes. (c) Box-tree representation.

Each FIGUE constructor (or combinator) is written in JAVA according to a predefined interface. Each combinator has its own formatting algorithm specifying the relative position of its children and determining the size and the alignment of its bounding box (the whole box). This algorithm updates the box's graphical properties (or attributes): origin, size, and alignment, by taking into account the display area parameters like the page width and the different parameters related to the graphical context of each box (when using multiple fonts for example). Once the formatting task is done, FIGUE does a single pass to display the formatted objects of the document according to their graphical contexts (font, color, background, coordinates) and draws the needed symbols or typographical characters (for example, the mathematic root symbol). Note that FIGUE has an incremental layout algorithm, with minimal re-drawing actions when data is modified, for a more efficient user-friendly result. FIGUE offers several default graphical constructors: *Paragraph, Horizontal, Vertical, Atomic*. To display mathematics we have added some new constructors such as *Root, Fraction, Table, Subscript, Supscript, Subsupscript*, etc., implementing the different methods dedicated to formatting, drawing, and allowing incremental redisplay. For this purpose, we have followed the MATHML-*presentation* recommendations to build our constructors with the same semantics and attributes.

FIGUE offers an effective selection mechanism and allows the direct manipulation of the underlying structured objects (computation, formula simplification, automatic command generation, ...) using mouse events.

FIGUE can be an excellent candidate for building WYSIWYG environments for mathematics. It is already in use at INRIA for building interactive tools and structured editors in order to manipulate objects like programs, mathematical formula or proof commands. As an example of application, it is used for developing of the PCOQ [ABPR01] system, a graphical interactive interface for the proof system COQ [HKPM97].

FIGUE also allows a module to render MathML. It can be used by any Java application like a MathML rendering interface. In the next section, we describes how to process and render MathML in FIGUE.

3.2 Processing and Rendering MathML in FIGUE

As we have followed MathML-*presentation* recommendations to design our mathematical constructors, the correspondence between MathML-*presentation* elements and FIGUE constructors is natural, which makes the translation easy. For each FIGUE graphical constructor, we have defined what the corresponding MathML-*presentation* element is.

Conversely, to be able to display MathML-*presentation* documents, we have developed in FIGUE a module to interpret the MathML-*presentation* format. This module uses a DOM-based (*Document Object Model*) generic XML parser to analyze and validate the MathML document and produces a DOM tree representing the MathML document. Next, this DOM tree is translated into a FIGUE box tree which can be displayed. This module establishes the link between the box structure and the MathML element, allowing thus the editing of MathML documents. To translate a MathML-*presentation* document into a FIGUE box tree, we make a direct translation, associating through some Java code each MathML element to a corresponding FIGUE constructor.

3.3 How to use FIGUE to display MathML presentation

FIGUE offers a module to handle MathML presentation. It possible to display the MathML presentation by this module and to interact with the displayed object. FIGUE can be used by any Java application which needs to handle MathML. In the following, we explain how to use FIGUE to display MathML presentation.

First, it is required that JDK1.3 is installed properly on the machine. Then, the XML parser (Xerces Java Parser 1.0.3) must be loaded.

The input should be a MathML File or a MathML String. The FIGUE MathML module will parse this MathML input and give the corresponding MathML Document (DOM Tree). Then, it will process and generate a FIGUE Facade (it contains box tree and the display resources) from this MathML document. The resulting FIGUE facade can eventually be displayed.

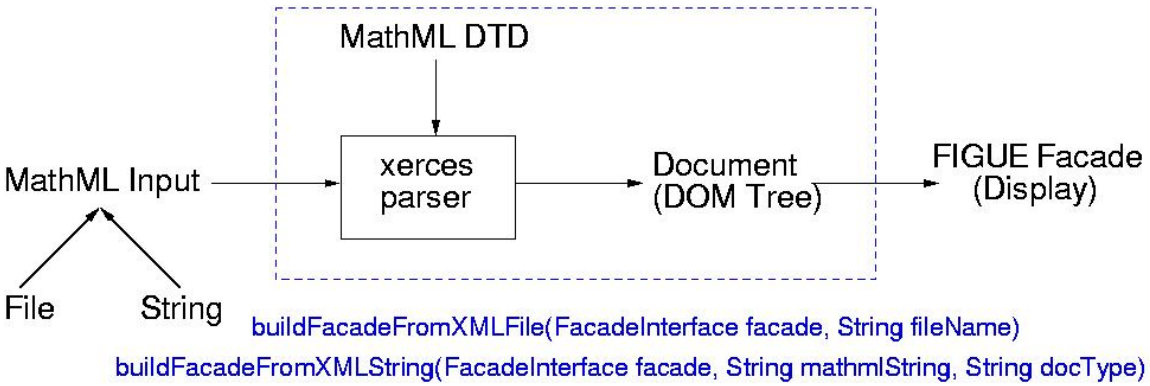


Figure 3: MathML process in FIGUE.

Once the FIGUE facade is displayed, it is possible to handle the displayed objects. One can select any object (structured selection) and get the corresponding box tree. The link between the FIGUE object structure and the corresponding MathML element is guaranteed. Whenever an object is selected, it is possible to get its corresponding MathML structure (see Figure 4).

The main methods of this MathML processing module are:

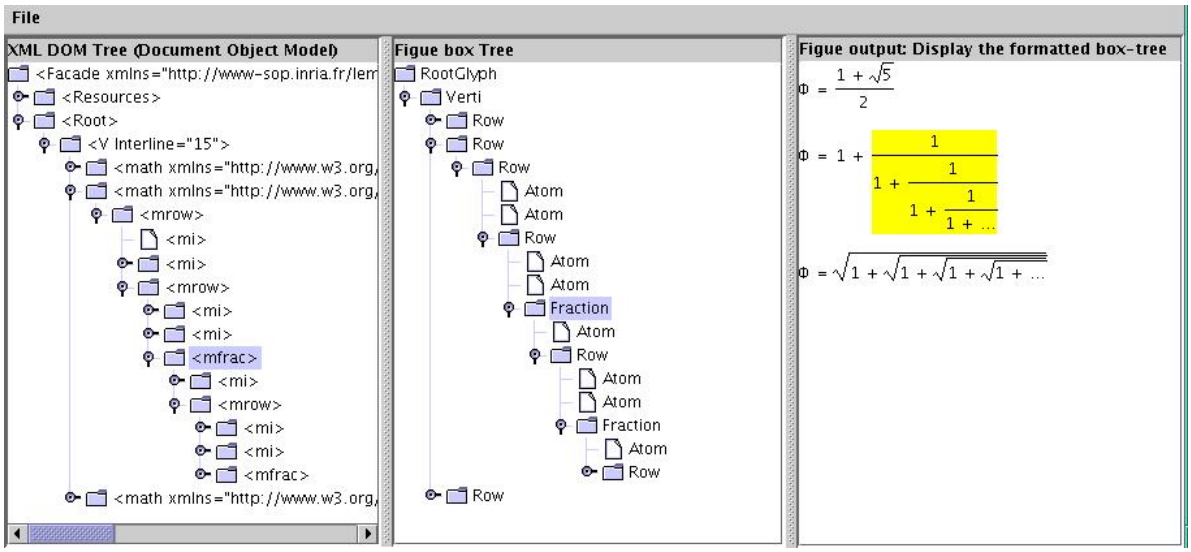


Figure 4: FIGUE gives the corresponding MathML structure of the selected object.

- **Document buildfacadeFromXMLFile(FacadeInterface facade, String fileName)**

  This method builds a FIGUE facade from an XML file. First, it parses the XML input and produces a corresponding document (DOM Tree) as its output. This document will be translated into a FIGUE box tree. This method has two important parts: a parser and a translator. The first part parses the MathML input with Xerces Java parser and the second part builds a FIGUE facade from the MathML document (it translates each MathML element in a FIGUE box tree).

- **Document buildFacadeFromXMLString(FacadeInterface facade, String mathmlString, String docType)**

  This method builds a FIGUE facade from an XML string. This method is like the previous method (see above). The difference is only at the parser level: this method parses the string input according to a given DTD (the doctype argument represents the path of the DTD in the DOCTYPE declaration) . This DTD is used to validate the XML input.

- **Document buildFacadeFromXMLString(FacadeInterface facade,String mathmlString)**

  It is about the same method (see above). Only the DOCTYPE declaration which specifies the DTD path is given with the string input. The input looks like:

```
<?xml version="1.0" encoding='UTF-8' standalone="no"?>
<!DOCTYPE math SYSTEM "/net/home/hnaciri/dtd/dtdMathML/mmlents/mathml.dtd">
<math xmlns="http://www.w3.org/1998/Math/MathML">
<mrow>
<mo>&lambda;</mo>
<mi>x</mi>
<mo>.</mo>
<mrow>
```

```
<mi>x</mi>
<mo>+</mo>
<mn>1</mn>
</mrow>
</mrow>
</math>
```

It is possible in FIGUE to select the displayed objects. The main methods to do the selection are:

- **GlyphInterface path2glyph(PathInterface path, GlyphInterface BoxeRoot)**

  This method returns the selected FIGUE box from the selection path. Usually this path is given from the selection event in FIGUE facade, this event translates a mouse click into a path in the box tree. The method also returns the root of the box tree representing all the document.

- **Node figue2xml(GlyphInterface selectedGlyph)**

  This method takes a selected box as its input and returns the corresponding MathML element. It uses a hash-table to obtain the MathML element corresponding to the box selected. The hash-table keeps a link between the box structure and the MathML structure.

- **StringWriter figue2xmlString(GlyphInterface selectedGlyph)**

  This method takes a selected box as its input and returns the corresponding XML serialization of the MathML element. This method also uses hash-table to obtain the MathML element corresponding to the box selected.

3.4 Example of MathML display by using FIGUE (Step by Step)

Here is an example of a program using this MathML module. The program loads a MathML file and displays it with FIGUEİt is possible to select the displayed object and to obtain the corresponding MathML element.

This Java program requires the FIGUE package, the Xerces Java parser package and other Java packages.

- **Import the usual figue packages**

```
import figue.*;
import figue.box.*;
import figue.resource.*;
import figue.io.xml.Utility;
```

- **Import the Xerces Java parser package**

```
import org.apache.xerces.parsers.DOMParser;
import org.xml.sax.SAXException;
import org.w3c.dom.Document;
```

- **Import the Java package**

```
import java.awt.*;
import java.awt.event.*;
import java.util.*;
import java.io.*;
import java.lang.reflect.*;
```

- **Type declarations**

  This is a FIGUE facade: `public  AWTFacade _facade;`

  The DOM tree will contain the parsing result of the MathML file:

```
public Document doc;
```

- **Main steps**

  - Create a FIGUE facade and initialize the resources for it (font, color ...)

```
_facade = new AWTFacade();
initResources();
```

  - Get the MathML file and parse it afterwards. Build a FIGUE facade from a MathML DOM tree (result of parser).

```
doc=Utility.buildFacadeFromXMLFile(_facade,filename);
```

  - Display a facade

```
_facade.buildInit(null);
_facade.buildComplete();
```

- **Selection part**

  We add a listener to the FIGUE facade. When a displayed object is selected, this program draws a box containing the selected object and returns the corresponding MathML structure.

```
_facade.addPointingListener(new PointingListenerInterface() {
  public final void mouseClicked(PointingEvent anEvent) {
    if ( anEvent.getPath() != null ) {
    // Here we draw a box including the selected object
    hightlightBox(anEvent.getPath());

    // Get the selected box structure
    GlyphInterface selectedGlyph =
    Utility.path2glyph(anEvent.getPath(),_facade.getRootGlyph());

    // From the selected box, we ask for the MathML element
    Node node_select= Utility.figue2xml(selectedGlyph);

    //From the selected box, we can ask directely for the corresponding
```

```
        //MathML text
        StringWriter xml_select = Utility.figue2xmlString(selectedGlyph);

          }
        }
      });
```

The source of the whole program in this example can be found at
http://www-sop.inria.fr/lemme/Hanane.Naciri/these/mathml/src/TestMathML.java

3.5 Sample application: Latex to HTML converter

The idea here is to make scientific documents written in Latex accessible on the Web and to be able to handle their structured objects like formulas. We are interested in producing an active Web version from a Latex document. For this purpose, we have chosen first to use a latex to XHTML+MathML converter and then convert the XHTML+MathML format to HTML+images. For each MathML formula, our program produces an image with specific areas and associates to each area its corresponding MathML structure (a sub formula). This association adds a structure to our images and allow us to handle the formulas.

Concerning the latex to XHTML+MathML converter, we have chosen to use WeM, a MathML editor that converts a subset of LaTeX to MathML (see
http://mathosphere.net/editeurml/WeM.html). To visualize the XHTML+MathML documents, different Web solutions exist: Web Browsers, applets, plug in .. We have chosen a simple solution which doesn't require to add any plugin nor to write transformation rules. We use our FIGUE MathML module (this module allows to render MathML formulas) to convert MathML formulas into images and to keep a link between the image and the MathML structure.

This module takes as argument the MathML formula and produces a corresponding image. This module computes the size of the formated formula and aligns it with the text. The resulting image is divided into several areas. For each area, we associate a MathML structure (to each MathML sub formula, we associate an area in the resulting image). The resulting document is an HTML document which includes some images and a subset of maps dividing each image into several areas. It is also possible to add a small JavaScript program in this document to get a structure of each selected formula (each area of each image).

The drawback of this solution is that the resulting document contains a lot of images when the latex source is big. The number of images is proportional to the number of formulas in the Latex source.

In order to experiment our program, we first used WeM to convert a latex source to an XHTML+MathML document. Then we used this file as an input of our program. The program converted this file into an HTML+images. The resulting document could be visualized by any Web browser. It was also possible to get the MathML structure of the displayed objects in each image.

References

[ABD+01] Ron Ausbrooks, Stephen Buswell, Stéphane Dalmas, Stan Devitt, and Angel Diaz et al. Mathematical Markup Language (MathML) Version 2.0 Specification. W3C Recommendation. `http://www.w3.org/TR/MathML2`, February 2001.

[ABD+02] Ron Ausbrooks, Stephen Buswell, Stéphane Dalmas, Stan Devitt, and Angel Diaz et al. Mathematical Markup Language (MathML) Version 2.0 (2nd Edition) Working Draft. `http://www.w3.org/TR/2002/WD-MathML2-20021219/`, December 2002.

[ABPR01] A. Amerkad, Y. Bertot, L. Pottier, and L. Rideau. Mathematics and proof presentation in pcoq. In *Proceedings of Workshop Proof Transformation and Presentation and Proof Complexities in connection with IJCAR 2001*, Siena, Italie, June 2001.

[CP02a] Paolo Casarini and Luca Padovani. Gnome DOM Engine. Web page, April 2002. `http://gdome2.cs.unibo.it`.

[CP02b] Paolo Casarini and Luca Padovani. The Gnome DOM Engine. *Markup Languages: Theory & Practice*, 3(2):173–190, April 2002.

[GTK] GTK+: the Gimp ToolKit. Web site. `http://www.gtk.org`.

[HHN+00] Arnaud Le Hors, Philippe Le Hégaret, Gavin Nicol, Jonathan Robie, and Mike Champion et al. (Eds). Document Object Model (DOM) Level 2 Core Specification Version 1.0. W3C Recommendation, November 2000. `http://www.w3.org/TR/DOM-Level-2-Core/`.

[HKPM97] G. Huet, G. Kahn, and C. Paulin-Mohring. The Coq Proof Assistant : A tutorial : Version 6.1. Technical Report RT-0204, INRIA, 1997.

[Knu90] D. Knuth. *The TeX-Book (revised).* Addison Wesley, 1190.

[Pad02] Luca Padovani. A Stand-Alone Rendering Engine for MathML. In Anonymous, editor, *MathML International Conference: Hickory Ridge Conference Center, Chicago, IL, USA, June 28–30*, 2002.