An Introduction to MetaUML

Exquisite UML Diagrams in MetaPost

Abstract
MetaUML is a GNU GPL MetaPost library for typesetting exquisite UML (Unified Modeling Language) diagrams. MetaUML offers a highly customizable, object-oriented API, designed with the ease of use in mind. This paper presents usage examples as well as a description of MetaUML infrastructure. This infrastructure may prove useful for general MetaPost typesetting, providing object-oriented replacements and enhancements to functionalities offered by the boxes package.

Keywords
MetaPost, TeX, LaTeX, UML, class diagram, state machine diagram, use case diagram, activity diagram

Introduction

Figure 1 presents a gallery of diagrams created by MetaUML (Gheorghies (2005)).

The code which generates these diagrams is quite straightforward, combining a natural object-oriented parlance with the power of MetaPost equation solving; for more information on MetaPost see Hobby (1992).

An UML class, for example, can be defined as follows:

```
Class.A("MyClass")
  ("attr1: int", "attr2: int")
  ("method1(): void", "method1(): void");
```

This piece of code creates an instance of `Class`, which will be afterward identified as `A`. This object has the following content properties: a name (`MyClass`), a list of attributes (`attr1`, `attr2`) and a list of methods (`method1`, `method2`). The one thing remaining before actually drawing `A` is to set its location:

```
A.nw = (0, 0);
drawObject(A);
```

In `A.nw` we refer to the “north-west” of the class rectangle, that is to its upper-left corner. In general, every MetaUML object has the positioning properties given in figure 2. These properties are used to set where to draw a given object, whether by assigning them absolute values, or by setting them relatively

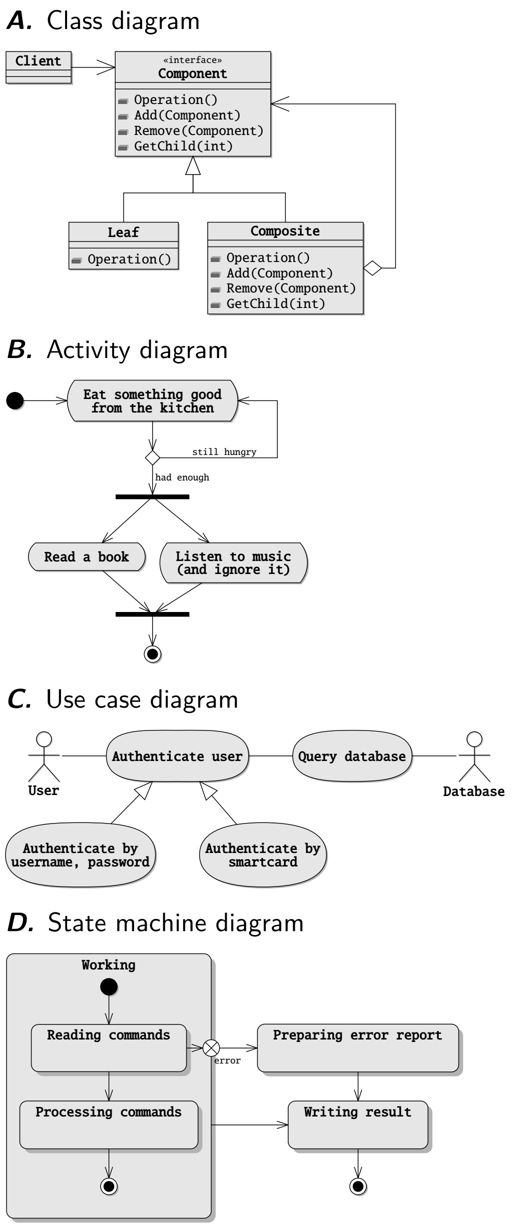


Figure 1. UML diagrams created by MetaUML.

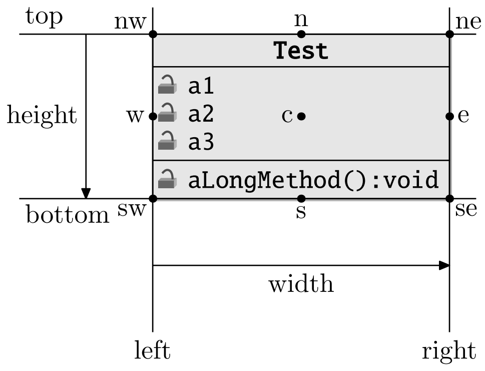


Figure 2. Positioning properties of any MetaUML object (here a class object is depicted).

to other objects. Suppose that we have defined two classes A and B. Then the following code would give a conceivable positioning:

```
A.nw = (0,0);
B.e = A.w + (-20, 0);
```

After the objects are drawn, one may draw links between them, such as inheritance or association relations between classes in class diagrams, or transitions between states in state machine diagrams. Whichever the purpose is, MetaUML provides a generic way of drawing an edge in a diagram's graph:

```
link(how-to-draw-information)(path-to-draw);
```

The "how to draw information" is actually an object which defines the style of the line (e.g. solid, dashed) and the appearance of the heads (e.g. nothing, arrow, diamond). One such object, called `inheritance`, defines a solid path ending in a white triangle. The `path-to-draw` parameter is simply a MetaPost path. For example, the following code can be used used to represent that class B is derived from A:

```
link(inheritance)(B.e -- A.w);
```

Note that the direction of the path is important, and MetaUML uses it to determine the type of adornment to attach at the link ends (if applicable). In our example, a white triangle, denoting inheritance, points towards the end of the path, that is towards class A.

To sum up, we present a short code and the resulting diagram (figure 3). This is typical for everything else in MetaUML. The positioning of A does not need to be explicitly set because "floating" objects are automatically positioned at (0,0) by their draw method.

```
input metauml;
beginfig(1);
  Class.A("A")()();
  Class.B("B")()();
  B.e = A.w + (-20, 0);
  drawObjects(A, B);
```

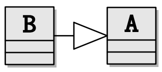


Figure 3. Example of MetaUML. Everything else works the same.

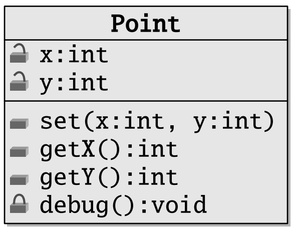


Figure 4. Class usage: name, attributes, methods and visibility markers.

```
  link(inheritance)(B.e -- A.w);
endfig;
end
```

From a user's perspective, this is all there is to MetaUML. With a reference describing how other UML elements are created, one can set out to typeset arbitrary complex diagrams.

Class Diagrams

A class is created as follows:

```
Class.name(class-name)
  (list-of-attributes)
  (list-of-methods);
```

The suffix `name` gives a name to the `Class` object (which, of course, represents an UML class). The name of the UML class is a string given by `class-name`; the attributes are given as a comma separated list of strings, `list-of-attributes`; the methods are given as a comma separated list of strings, `list-of-attributes`. The list of attributes and the list of methods may be void.

Each of the strings representing an attribute or a method may begin with a visibility marker: "+" for public, "#" for protected and "−" for private. MetaUML interprets this marker and renders a graphic stereotype in form of a lock which may be opened, semi-closed and closed, respectively.

The following code yields the diagram in figure 4.

```
Class.A("Point")
  ("#x:int", "#y:int")
  ("+set(x:int, y:int)",
   "+getX():int",
   "+getY():int",
   "-debug():void");
drawObject(A);
```

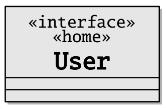


Figure 5. Class usage: stereotypes.

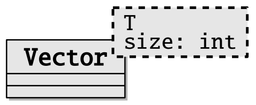


Figure 6. Class usage: templates.

Stereotypes

After a class is created, its stereotypes may be specified by using the macro `classStereotypes`:

```
classStereotypes.name(list-of-stereotypes);
```

Here, `name` is the object name of a previously created class and `list-of-stereotypes` is a comma separated list of strings. Here is an example along with the resulting diagram (figure 5).

```
Class.A("User")();
classStereotypes.A("<<interface>>", "<<home>>");

drawObject(A);
```

Parametrized Classes (Templates)

The most convenient way of typesetting a class template in MetaUML is to use the macro `ClassTemplate`. This macro creates a visual object which is appropriately positioned near the class object it adorns.

```
ClassTemplate.name(list-of-templates)
                  (class-object);
```

The `name` is the name of the template object, `list-of-templates` is a comma separated list of strings and the `class-object` is the name of a class object.

The code below results in the diagram from figure 6.

```
Class.A("Vector")();
ClassTemplate.T("T", "size: int")(A);

drawObjects(A, T);
```

The macro `Template` can also be used to create a template object, but this time the resulting object can be positioned freely.

```
Template.name(list-of-templates);
```

Of course, one can specify both stereotypes and template parameters for a given class.

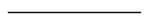
Types of Links

In this section we enumerate the relations that can be drawn between classes by means of MetaUML macros.

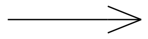
Suppose that we have the declared two points, `A` (on the left) and `B` (on the right):

```
pair A, B;
A = (0,0);
B = (50,0);
```

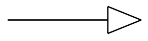
```
Bidirectional association.
link(association)( A -- B );
```



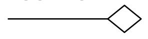
```
Unidirectional association.
link(associationUni)( A -- B );
```



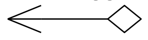
```
Inheritance. link(inheritance)( A -- B );
```



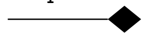
```
Aggregation. link(aggregation)( A -- B );
```



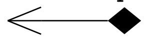
```
Unidirectional aggregation.
link(aggregationUni)( A -- B );
```



```
Composition. link(composition)( A -- B );
```



```
Unidirectional composition.
link(compositionUni)( A -- B );
```



Associations

In UML an association typically has two of association ends and may have a name specified for it. In turn, each association end may specify a multiplicity, a role, a visibility, an ordering. These entities are treated in MetaUML as pictures having specific drawing information (spacings, font).

The first method of creating association "items" is by giving them explicit names. Having a name for an association item comes in handy when referring to its properties is later needed (see the non UML-compliant diagram in figure 7). Note that the last parameter of the macro `item` is an equation which uses the item name to perform positioning.

```
Class.P("Person")();
Class.C("Company")(); % drawing code ommited

item.aName(iAssoc)("works for")
        (aName.s = .5[P.w, C.w]);
draw aName.n -- (aName.n + (20,20));
label.urt("association name" infont "tyxtt",
        aName.n + (20,20));
```

However, giving names to every association item may become an annoying burden (especially when

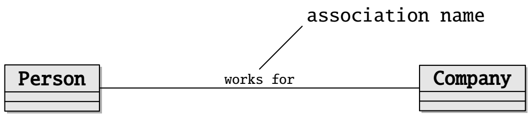


Figure 7. Referring to the properties of association items.

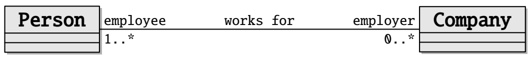


Figure 8. Anonymous association items.

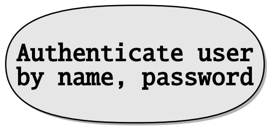


Figure 9. Usecase example.

there are many of them). Because of this, MetaUML also allows for "anonymous items". In this case, the positioning is set by an equation which refers to the anonymous item as `obj` (figure 8).

```
% P and C defined as in the previous example

item(iAssoc)("employee")(obj.sw = P.e);
item(iAssoc)("1..*")(obj.nw = P.e);

% other items are drawn similarly
```

Use Case Diagrams

Use Cases

An use case is created by the macro `Usecase`:

```
Usecase.name(list-of-lines);
```

The `list-of-lines` is a comma separated list of strings. These strings are placed on top of each other, centered and surrounded by the appropriate visual UML notation.

Use case example (result in figure 9):

```
Usecase.U("Authenticate user",
          "by name, password");
drawObject(U);
```

Actors

An actor is created by the macro `Actor`:

```
Actor.name(list-of-lines);
```

Here, `list-of-lines` represents the actor's name. For convenience, the name may be given as a list of strings which are placed on top of each other, to provide support for the situations when the role is quite long. Otherwise, giving a single string as an argument to the Actor constructor is perfectly fine.

Actor example (result in figure 10):

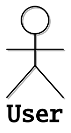


Figure 10. Actor example.

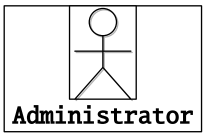


Figure 11. Actor example, accessing the "human".

```
Actor.A("User");
drawObject(A);
```

Note that one may prefer to draw diagram relations positioned relatively to the visual representation of an actor (the "human") rather than relatively to the whole actor object (which also includes the text). Because of that, MetaUML provides access to the "human" of every actor object `actor` by means of the sub-object `actor.human`. Figure 11 gives the result of the code below:

```
Actor.A("Administrator");
drawObject(A);
draw objectBox(A);
draw objectBox(A.human);
```

Note that in MetaUML `objectBox(X)` is equivalent to `X.nw -- X.ne -- X.se -- X.sw -- cycle` for every object `X`.

Types of Links

Some of the types of links defined for class diagrams (such as inheritance, association etc.) can be used with similar semantics within use case diagrams.

Activity Diagrams

Begin and End

The begin and the end of an activity diagram can be marked by using the macros `Begin` and `End`, respectively. The constructors of these visual objects take no parameters:

```
Begin.beginName;
End.endName;
```

Figure 12 gives the output of the code:

```
Begin.b;
End.e;
b.nw = (0,0);
e.nw = (20, 20);

drawObjects(b, e);
```

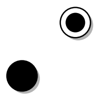


Figure 12. Begin and end in an activity diagram.

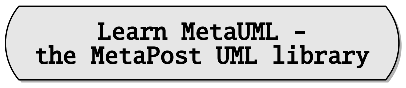


Figure 13. Activity example.

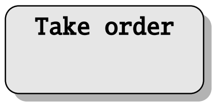


Figure 14. State example.

Activity

An activity is constructed as follows:

```
Activity.name(list-of-strings);
```

The parameter `list-of-strings` is a comma separated list of strings. These strings are centered on top of each other to allow for the accommodation of a longer activity description within a reasonable space.

An example is given in figure 13

```
Activity.A("Learn MetaUML -",
           "the MetaPost UML library");
drawObject(A);
```

Types of Links

In activity diagrams, transitions between activities are needed. They are typeset as in the example below. Figure 15 shows such a transition rendered. This type of link is also used for state machine diagrams.

```
link(transition)( pointA -- pointB );
```

State Diagrams

The constructor of a state allows for aggregated substates:

```
State.name(state-name)(substates-list);
```

The parameter `state-name` is a string or a list of comma separated strings representing the state's name or description. The `substates-list` parameter is used to specify the substates of this state as a comma separated list of objects; this list may be void.

Figure 14 presents a simple state, rendered by the following code:

```
State.s("Take order")();
drawObject(s);
```

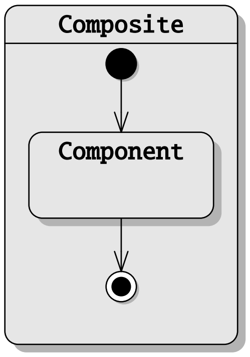


Figure 15. State example: composite states.

Composite States

A composite state is defined by enumerating at the end of its constructor the inner states. Interestingly enough, the composite state takes care of drawing the sub-states it contains. The transitions must be drawn after the composite state, as seen in the next example (figure 15):

```
Begin.b;
End.e;
State.c("Component")();
State.composite("Composite")(b, e, c);

b.midx = e.midx = c.midx;
c.top = b.bottom - 20;
e.top = c.bottom - 20;

composite.info.drawNameLine := 1;
drawObject(composite);

link(transition)(b.s -- c.n);
link(transition)(c.s -- e.n);
```

Internal Transitions

Internal transitions can be specified by using the macro:

```
stateTransitions.name(list-transitions);
```

Identifier `name` gives the state object whose internal transitions are being set, and parameter `list-transitions` is a comma separated string list. Figure 16 presents the result of the code below.

```
State.s("An interesting state",
        "which is worth mentioning")();
stateTransitions.s(
  "OnEntry / Open eyes",
  "OnExit  / Sleep well");
s.info.drawNameLine := 1;

drawObject(s);
```

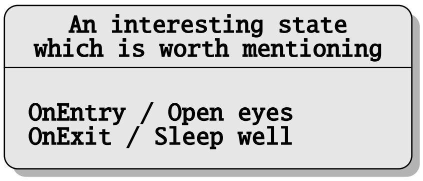


Figure 16. State example: internal transitions.

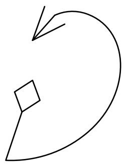


Figure 17. Link paths can be arbitrary complex in MetaUML: the heads are properly drawn.

Special States
Similarly to the usage of `Begin` and `End` macros, one can define history states, exit/entry point states and terminate pseudo-states, by using the following constructors.

```
History.nameA;
ExitPoint.nameB;
EntryPoint.nameC;
Terminate.nameD;
```

Drawing Paths

The `link` macro is powerful enough to draw relations following arbitrary paths (figure 17):

```
za = (10,10);
zb = (80,-10);
path cool;
cool := za .. za+(20,10) ..
        zb+(20,-40) ..
        zb+(-10,-30) -- zb;
link(aggregationUni)(cool);
```

Regardless of how amusing this feature might be, it does become a bit of a nuisance to use it in its bare form. When typesetting UML diagrams in good style, one generally uses rectangular paths. It is for this kind of style that MetaUML offers extensive support, providing a “syntactic sugar” for constructs which can otherwise be done by hand, but with some extra effort.

Manhattan Paths
The “Manhattan” path macros generate a path between two points consisting of one horizontal and one vertical segment. The macro `pathManhattanX` generates first a horizontal segment, while the macro `pathManhattanY` generates first a vertical segment. In MetaUML it also matters the direction of a path, so you

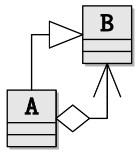


Figure 18. Manhattan paths.

can choose to reverse it by using `rpathManhattanX` and `rpathManhattanY` (note the prefix “r”):

```
pathManhattanX(A, B)
pathManhattanY(A, B)

rpathManhattanX(A, B)
rpathManhattanY(A, B)
```

Figure 18 shows these macros at work:

```
Class.A("A")()();
Class.B("B")()();

B.sw = A.ne + (10,10);
drawObjects(A, B);

link(aggregationUni)
   (rpathManhattanX(A.e, B.s));
link(inheritance)
   (pathManhattanY(A.n, B.w));
```

Stair Step Paths
These path macros generate stair-like paths between two points. The “stair” can “rise” first in the direction of Ox axis (`pathStepX`) or in the direction of Oy axis (`pathStepY`). How much should a step rise is given by an additional parameter, `delta`. Again, the macros prefixed with “r” reverse the direction of the path given by their unprefixed counterparts.

```
pathStepX(A, B, delta)
pathStepY(A, B, delta)

rpathStepX(A, B, delta)
rpathStepY(A, B, delta)
```

Figure 19 shows these macros at work:

```
stepX:=60;
link(aggregationUni)
   (pathStepX(A.e, B.e, stepX));

stepY:=20;
link(inheritance)
   (pathStepY(B.n, A.n, stepY));
```

Horizontal and Vertical Paths
There are times when drawing horizontal or vertical links is required, even when the objects are not properly aligned. To this aim, the following macros are use-

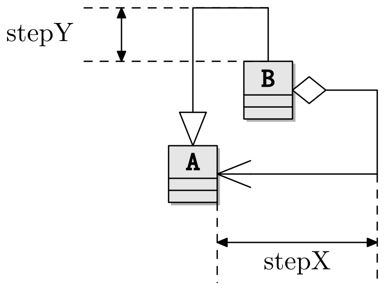


Figure 19. Stair step paths.

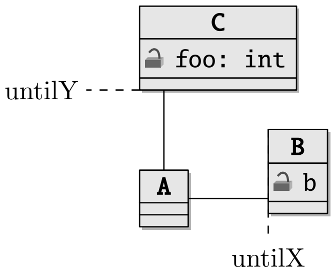


Figure 20. Horizontal and vertical paths.

ful:

```
pathHorizontal(pA, untilX)
pathVertical(pA, untilY)

rpathHorizontal(pA, untilX)
rpathVertical(pA, untilY)
```

A path created by `pathHorizonal` starts from the point `pA` and continues horizontally until coordinate `untilX` is reached. The macro `pathVertical` constructs the path dually, working vertically. The prefix "`r`" reverses the direction of the path.

Figure 20 gives an usage example:

```
untilX := B.left;
link(association)
   (pathHorizontal(A.e, untilX));

untilY:= C.bottom;
link(association)
   (pathVertical(A.n, untilY));
```

Direct Paths

A direct path can be created with `directPath`. The call `directPath(A, B)` is equivalent to `A -- B`.

Paths between Objects

Using the constructs presented above, it is clear that one can draw links between diagram objects, using a code like:

```
link(transition)(directPath(objA.nw, objB.se));
```

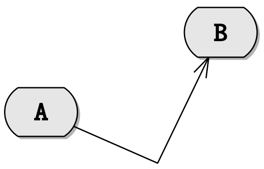


Figure 21. The `pathCut` macro at work.

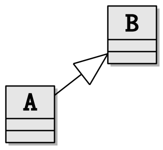


Figure 22. Direct linking between objects with `clink`.

There are times however this may yield unsatisfactory visual results, especially when the appearance of the object's corners is round. MetaUML provides the macro `pathCut` whose aim is to limit a given path exactly to the region outside the actual borders of the objects it connects. The macro's synopsis is:

```
pathCut(thePath)(objectA, objectB)
```

Here, `thePath` is a given MetaPost path and `objectA` and `objectB` are two MetaUML objects. By contract, each MetaUML object of type, say, `X` defines a macro `X_border` which returns the path that surrounds the object. Because of that, `pathCut` can make the appropriate modifications to `thePath`.

The following code demonstrates the benefits of the `pathCut` macro (figure 21):

```
z = A.se + (30, -10);
link(transition)
   (pathCut(A, B)(A.c--z--B.c));
```

Direct Paths between Centers. At times is quicker to just draw direct paths between the center of two objects, minding of course the object margins. The macro which does this is `clink`:

```
clink(how-to-draw-information)(objA, objB);
```

The parameter `how-to-draw-information` is the same as for the macro `link`; `objA` and `objB` are two MetaUML objects.

Figure 22 gives the output of the following code:

```
  clink(inheritance)(A, B);
```

The MetaUML Infrastructure

MetaPost is a macro language based on equation solving. Using it may seem quite tricky at first for a programmer accustomed to modern object-oriented languages. However, the great power of MetaPost consists in its versatility. Indeed, it is possible to write a

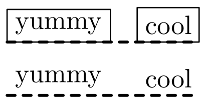


Figure 23. Motivation for not using boxes: the bottom alignment is imperfect.

system which mimics quite well object-oriented behavior. Along this line, METAOBJ (Roegel (2002)) is a library worth mentioning: it provides a high-level objects infrastructure along with a battery of predefined objects.

Surprisingly enough, MetaUML does not use METAOBJ. Instead it uses a custom written, lightweight object-oriented infrastructure, provisionally called "util". The fact that METAOBJ's source consists of a huge file which is rather hard to follow and understand contributed to this decision.

Another library that has some object-oriented traits is the boxes library, which comes with the standard MetaPost distribution. Early versions of MetaUML did use boxes as an infrastructure, but eventually it had to be abandoned. The main reason was that it was difficult to achieve good visual results when stacking texts (more on that further on). Also, it had a degree of flexibility which became apparent to be insufficient.

Motivation

Suppose that we want to typeset two texts with their bottom lines aligned, using boxit (figure 23):

```
boxit.a ("yummy");
boxit.b ("cool");

a.nw = (0,0); b.sw = a.se + (10,0);

drawboxed (a, b); % or drawunboxed(a,b)
draw a.sw -- b.se dashed evenly
   withpen pencircle scaled 1.1;
```

Note that "yummy" *looks* slightly higher than "cool": this is unacceptable when, in an UML class diagram, roles are placed at the ends of a horizontal association. Regardless of default spacing being smaller in the util library, the very same unfortunate misalignment effect rears its ugly head (figure 24):

```
Picture.a("yummy");
Picture.b("cool");
% comment next line for unboxed
a.info.boxed := b.info.boxed := 1;

b.sw = a.se + (10,0);

drawObjects(a, b);
```

However, the strong point of util is that we have a recourse to this problem (figure 25):

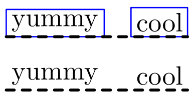


Figure 24. Misalignment occurs by default with the util library, but this can be configured not to happen.

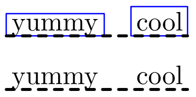


Figure 25. The util package provides good alignment.

```
iPict.ignoreNegativeBase := 1;

Picture.a("yummy");
Picture.b("cool");
% the rest the same as above
drawObjects(a, b);
```

The Picture Macro

We have seen previously the line iPict.ignoreNegativeBase := 1. Who is iPict and what is it doing in our program? MetaUML aims at separating the "business logic" (what to draw) from the "interface" (how to draw). In order to achieve this, it records the "how to draw" information within the so-called Info structures. The object iPict is an instance of PictureInfo structure, which has the following properties (or attributes):

```
left, right, top, bottom
ignoreNegativeBase
boxed, borderColor
```

The first four attributes specify how much space should be left around the actual item to be drawn. The marvelous effect of ignoreNegativeBase has just been shown (off) while the last two attributes control whether the border should be drawn (when boxed=1) and if drawn, in which color.

There's one more thing: the font to typeset the text in. This is specified in a FontInfo structure which has two attributes: the font name and the font scale. This information is kept within the PictureInfo structure as a contained attribute iFont. Both FontInfo and PictureInfo have "copy constructors" which can be used to make copies. We have already the effect of these copy constructors at work, when we used:

```
Picture.a("yummy");
a.info.boxed := 1;
```

A copy of the default info for a picture, iPict, has been made within the object a and can be accessed as a.info. Having a copy of the info in each object may seem like an overkill, but it allows for a fine grained control of the drawing mode of each individual object.

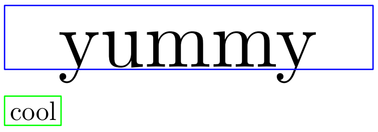


Figure 26. Having predefined configurations yields short, convenient code.

This feature comes in very handy when working with a large number of settings, as it is the case for MetaUML.

Let us imagine for a moment that we have two types of text to write: one with a small font and a small margin and one with a big font and a big margin. We could in theory configure each individual object or set back and forth global parameters, but this is far for convenient. It is preferable to have two sets of settings and specify them explicitly when they are needed. The following code could be placed somewhere in a configuration file and loaded before any `beginfig` macro:

```
PictureInfoCopy.iBig(iPict);
iBig.left := iBig.right := 20;
iBig.top := 10;
iBig.bottom := 1;
iBig.boxed := 1;
iBig.ignoreNegativeBase := 1;
iBig.iFont.name := defaultfont;
iBig.iFont.scale := 3;

PictureInfoCopy.iSmall(iPict);
iSmall.boxed := 1;
iSmall.borderColor := green;
```

Below is an usage example of these definitions (result in figure 26). Note the name of the macro: `EPicture`. The prefix comes form “explicit” and it’s used to acknowledge that the “how to draw” information is set explicitly, as opposed to the `Picture` macro where the `info` member defaults to `iPict`.

```
EPicture.a(iBig)("yummy");
EPicture.b(iSmall)("cool");
% you can still modify a.info and b.info

b.sw = a.se + (10,0);

drawObjects(a, b);
```

Stacking Objects

It is possible to stack objects, much in the style of `setboxjoin` from boxes library (figure 27).

```
Picture.a0("yummy");
Picture.a1("cool");
Picture.a2("fool");

setObjectJoin(pa.sw = pb.nw);
```

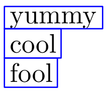


Figure 27. Stacking objects.

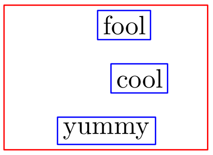


Figure 28. Grouping objects.

```
joinObjects(scantokens listArray(a)(3));
drawObjects(scantokens listArray(a)(3));
% or drawObjects (a0, a1, a2);
```

The `listArray` macro provides here a shortcut for writing `a0, a1, a2`. This macro is particularly useful for generic code which does not know beforehand the number of elements to be drawn. Having to write the `scantokens` keyword is admittedly a nuisance, but this is required.

The Group Macro

It is possible to group objects in MetaUML. This feature is the cornerstone of MetaUML, allowing for the easy development of complex objects, such as composite stats in state machine diagrams.

Similarly to the macro `Picture`, the structure `GroupInfo` is used for specifying group properties; its default instantiation is `iGroup`. Furthermore, the macro EGroup explicitely sets the layout information. Figure 28 results from the code below:

```
iGroup.left:=20;
iGroup.right:=15;
iGroup.boxed:=1;
iPicture.boxed:=1;

Picture.a("yummy");
Picture.b("cool");
Picture.c("fool");

b.nw = a.nw + (20,20);  % A
c.nw = a.nw + (15, 40); % B

Group.g(a, b, c);
g.nw = (10,10); % C

drawObject(g);
```

Note that after some objects are grouped, they can all be drawn by invoking the `drawObject` macro solely on the group that aggregates them. Another important remark is that it is necessary only to set the relative positioning of objects within a group (line A and

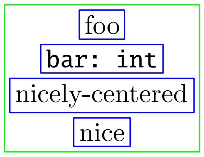


Figure 29. An example of a picture stack.

B); afterward, one can simply "move" the group to a given position (line C), and all the contained objects will move along.

The PictureStack Macro

The `PictureStack` macro is a syntactic sugar for a set of pictures, stacked according to predefined equations and grouped together (figure 29).

```
iStack.boxed := 1;
iStack.iPict.boxed := 1;
PictureStack.myStack("foo",
  "bar: int" infont "tyxtt",
  "nicely-centered" infont defaultfont,
  "nice")("vcenter");

drawObject(myStack);
```

Note the last parameter of the macro `PictureStack`, here `vcenter`. It is used to generate appropriate equations based on a descriptive name. The spacing between individual picture objects is set by the field `iStack.spacing`. Currently, the following alignment names are defined: `vleft`, `vright`, `vcenter`, `vleftbase`, `vrightbase`, `vcenterbase`. All these names refer to vertical alignment (the prefix "v"); alignment can be at left, right or centered. The variants having the suffix "base" align the pictures so that `iStack.spacing` refer to the distance between the bottom lines of the pictures. The unsuffixed variants use `iStack.spacing` as the distance between one's bottom line and the next's top line.

The "base" alignment is particularly useful for stacking text, since it offers better visual appearance when `iPict.ignoreNegativeBase` is set to 1.

Components Design

Each MetaUML component (e.g. `Picture`, `PictureStack`, `Class`) is designed according to an established pattern. This section gives more insight on this.

In order to draw a component, one must know the following information:

- □ what to draw, or what are the elements of a component.
- □ how to draw, or how are the elements positioned in relation to each other within the component
- □ where to draw

For example, in order to draw a picture object we must know, respectively:

- □ what is the text or the native picture that needs to be drawn
- □ what are the margins that should be left around the contents
- □ where is the picture to be drawn

Why do we bother with these questions? Why don't we just simply draw the picture component as soon as it was created and get it over with? That is, why doesn't the following code just work?

```
Picture.pict("foo");
```

Well, although we have the answer to question 1 (what to draw), we still need to have question 3 answered. The code below becomes thus a necessity (actually, you are not forced to specify the positioning of an object, because its draw method positions it to `(0,0)` by default):

```
% question 1: what to draw
Picture.pict("foo");

% question 3: where to draw
pict.nw = (10,10);

% now we can draw
drawObject(pict);
```

How about question 2, how to draw? By default, this problem is addressed behind the scenes by the component. This means, for the Picture object, that a native picture is created from the given string, and around that picture certain margins are placed, by means of MetaPost equations. (The margins come in handy when one wants to quickly place Picture objects near others, so that the result doesn't look too cluttered.) If these equations were defined within the Picture constructor, then an usability problem would have appeared, because it wouldn't have been possible to modify the margins, as in the code below:

```
% question 1: what to draw
Picture.pict("foo");

% question 2: how to draw
pict.info.left := 10;
pict.info.boxed := 1;

% question 3: where to draw
pict.nw = (0,0);
```

```
% now we can draw
drawObject(pict);
```

To allow for this type of code, the equations that define the layout of the `Picture` object (here, what the margins are) must be defined somewhere after the constructor. This is done by a macro called `Picture_layout`. This macro defines all the equations which link the "what to draw" information to the "how to draw" information (which in our case is taken from the `info` member, a copy of `iPict`). Nevertheless, notice that `Picture_layouts` is not explicitly invoked. To the user's great relief, this is taken care of automatically within the `Picture_draw` macro.

There are times however, when explicitly invoking a macro like `Picture_layout` becomes a necessity. This is because, by contract, it is only after the `layout` macro is invoked that the final dimensions (width, height) of an object are definitely and permanently known. Imagine that we have a component whose job is to surround in a red-filled rectangle some other objects. This component needs to know what the dimensions of the contained objects are, in order to be able to set its own dimensions. At drawing time, the contained objects must not have been drawn already, because the red rectangle of the container would overwrite them. Therefore, the whole pseudo-code would be:

```
Create objects o1, o2, ... ok;
Create container c(o1, o2, ..., ok);
Optional: modify info-s for o1, o2, ... ok;
Optional: modify info for c;

layout c, requiring layout of o1, o2, ... ok;
establish where to draw c;
draw red rectangle defined by c;
draw components o1, o2, ...ok within c
```

Note that an object mustn't be laid out more than once, because otherwise inconsistent or superfluous equations would arise. To enforce this, by contract, any object must keep record of whether its layout method has already been invoked, and if the answer is affirmative, subsequent invocations of the layout macro would do nothing. It is very important to mention that after the `layout` macro is invoked over an object, modifying the `info` member of that object has no subsequent effect, since the layout equations are declared and interpreted only once.

Notes on the Implementation of Links

MetaUML considers edges in diagram graphs as links. A link is composed of a path and the heads (possible none, one or two). For example, an association has no heads, and one must simply draw along the path with a solid pen. An unidirectional aggregation has a solid path and two heads: one is an arrow and the other is a diamond. So the template algorithm for drawing a link is:

AB	the path specified by the user
$\lvert AA'\rvert$	`iLink.widthA`
$\lvert BB'\rvert$	`iLink.widthB`

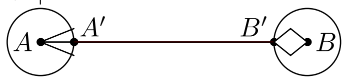


Figure 30. Details on how a link is drawn by MetaUML.

```
0. Reserve space for heads
1. Draw the path (except for the heads)
2. Draw head 1
3. Draw head 2
```

Each of the UML link types define how the drawing should be done, in each of the cases (1, 2 and 3). Consider the link type of unidirectional composition. Its "class" is declared as:

```
vardef CompositionUniInfo@# =
  LinkInfo@#;

  @#widthA      = defaultRelationHeadWidth;
  @#heightA     = defaultRelationHeadHeight;
  @#drawMethodA = "drawArrow";

  @#widthB      = defaultRelationHeadWidth;
  @#heightB     = defaultRelationHeadHeight;
  @#drawMethodB = "drawDiamondBlack";

  @#drawMethod = "drawLine";
enddef;
```

Using this definition, the actual description is created like this:

```
CompositionUniInfo.compositionUni;
```

As shown previously, is is the macro `link` which performs the actual drawing, using the link description information which is given as parameter (generally called `iLink`). For example, we can use:

```
link(aggregationUni)((0,0)--(40,0));
```

Let us see now the inner workings of macro `link`. Its definition is:

```
vardef link(text iLink)(expr myPath)=
   LinkStructure.ls(myPath,
                    iLink.widthA, iLink.widthB);
   drawLinkStructure(ls)(iLink);
enddef;
```

First, space is reserved for heads, by "shortening" the given path `myPath` by `iLink.widthA` at the beginning and by `iLink.widthB` at the end. After that, the shortened path is drawn with the "method" given by `iLink.drawMethod` and the heads with the "methods" `iLink.drawMethodA` and `iLink.drawMethodB`, respectively (figure 30).

Object Definitions: Easier `generic_declare`
In MetaPost if somebody wants to define something resembling a class, say `Person`, he would do something like this:

```
vardef Person@#(expr _name, _age)=
  % @# prefix can be seen as ‘this‘ pointer
  string @#name;
  numeric @#age;

  @#name := _name;
  @#age := _age;
enddef;
```

This allows for the creation of instances (or objects) of class `Person` by using declarations like:

```
Person.personA;
Person.personB;
```

However, if one also wants to able able to create indexed arrays of persons, such as `Person.student0`, `Person.student1` etc., the definition of class `Person` must read:

```
vardef Person@#(expr _name, _age)=
  _n_ := str @#;
  generic_declare(string) _n.name;
  generic_declare(numeric) _n.age;

  @#name := _name;
  @#age := _age;
enddef;
```

This construction is rather inelegant. MetaUML offers alternative macros to achieve the same effect, uncluttering the code by removing the need for the unaesthetic `_n_` and `_n`.

```
vardef Person@#(expr _name, _age)=
  attributes(@#);
  var(string) name;
  var(numeric) age;

  @#name := _name;
  @#age := _age;
enddef;
```

Customization in MetaUML: Examples

We have seen that in MetaUML the “how to draw” information is memorized into the so-called “Info” structures. For example, the default way in which a `Picture` object is to be drawn is recorded into an instance of `PictureInfo`, named `iPict`. In this section we present a case study involving the customization of `Class` objects. The customization of any other MetaUML objects works similarly. Here we can-

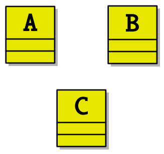


Figure 31. Changing the default settings for all classes.

not possibly present all the customization options for all kinds of MetaUML objects: this would take too long. Nevertheless, an interested reader can refer to the top of the appropriate MetaUML library file, where `Info` structures are defined. For example, class diagram related definitions are in `metauml_class.mp`, activity diagram definitions are in `metauml_activity.mp` etc.

Global settings

Let us assume that we do not particularly like the default foreground color of all classes, and wish to change it so something yellowish. In this scenario, one would most likely want to change the appropriate field in `iClass`:

```
iClass.foreColor := (.9, .9, 0);
```

After this, the following code produces the result in figure 31:

```
Class.A("A")()();
Class.B("B")()();
Class.C("C")()();

B.w = A.e + (20,0);
C.n = .5[A.se, B.sw] + (0, -10);

drawObjects(A, B, C);
```

Individual settings

When one wants to make modifications to the settings of one particular `Class` objects, another strategy is more appropriate. How about having class `C` stand out with a light blue foreground color, a bigger font size for the class name and a blue border (figure 32)?

```
iPict.foreColor := (.9, .9, 0);

Class.A("A")()();
Class.B("B")()();
Class.C("C")()();
C.info.foreColor := (.9, .7, .7);
C.info.borderColor := green;
C.info.iName.iFont.scale := 2;

% positioning code ommited
drawObjects(A, B, C);
```

As an aside, note that for each `Class` object its `info` member is created as a copy of `iClass`: the actual

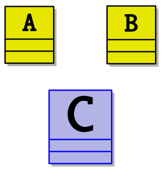


Figure 32. Individual customization of a class object.

drawing is performed using this copied information. Because of that, one can modify the `info` member after the object has been created and still get the desired results.

Another thing worth mentioning is that the `ClassInfo` structure contains the `iName` member, which is an instance of `PictureInfo`. In our example we do not want to modify the spacings around the `Picture` object, but the characteristics of the font its contents is typeset into. To do that, we modify the `iName.iFont` member, which by default is a copy of `iFont` (an instance of `FontInfo`, defined in `util_picture.mp`). If, for example, we want to change the font the class name is rendered into, we would set the attribute `iName.iFont.name` to a string representing a font name on our system (as used with the MetaPost `infont` operator).

Predefined settings

The third usage scenario is perhaps more interesting. Suppose that we have two types of classes which we want to draw differently. Making the setting adjustments for each individual class object would soon become a nuisance. MetaUML's solution consists in the ability of using predefined "how to draw" `Info` objects. Let us create such objects:

```
ClassInfoCopy.iHome(iClass);
iHome.foreColor := (0, .9, .9);

ClassInfo.iRemote;
iRemote.foreColor := (.9, .9, 0);
iRemote.borderColor := green;
```

Object `iHome` is a copy of `iClass` (as it might have been set at the time of the macro call). Object `iRemote` is created just as `iClass` is originally created. We can now use these `Info` objects to easily set the "how to draw" information for classes. The result is depicted in figure 33, please note the "E" prefix in `EClass`:

```
EClass.A(iHome)("UserHome")()();
EClass.B(iRemote)("UserRemote")()();
EClass.C(iHome)("CartHome")()();
EClass.D(iRemote)("CartRemote")()();
```

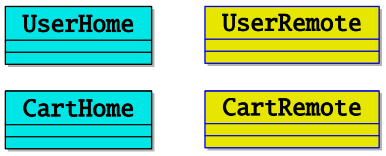


Figure 33. Using predefined settings.

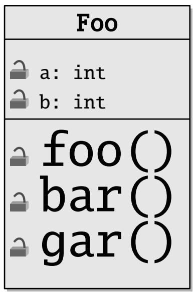


Figure 34. Extreme customization of a class. You may want not to do this, after all.

Extreme customization

When another font (or font size) is used, one may also want to modify the spacings between the attributes' and methods' baselines. Figure 34 is the result of the (unlikely) code:

```
Class.A("Foo")
  ("a: int", "b: int")
  ("foo()", "bar()", "gar()");
A.info.iAttributeStack.iPict.iFont.scale := 0.8;
A.info.iAttributeStack.top := 10;
A.info.iAttributeStack.spacing := 11;

A.info.iMethodStack.iPict.iFont.scale := 2;
A.info.iMethodStack.spacing := 17;
A.info.iMethodStack.bottom := 10;

drawObject(A);
```

Both `iAttributeStack` and `iMethodStack` are instances of `PictureStackInfo`, which is used to control the display of `PictureStack` objects.

Conclusions

MetaUML is a GNU GPL library for typesetting UML diagrams, particularly useful in a TeX or LaTeX environment; see Knuth (1986), Lamport (1994). It provides an easy to use, human readable API.

The code of a diagram typeset in MetaUML appears clearer (at least to the author of this paper) than the corresponding code in `uml.sty`, `pst-uml.sty`, `umldoc` or even XMI; see Gjelstad (2001), Diamantini (1998), Palmer (1999), OMG (2003). It is the next best thing to using a visual tool, while having the freedom of not becoming technologically dependent of any particular visual tool.

The `util` infrastructure of MetaUML offers means of defining and using "objects", which may recommend it for other typesetting projects, unrelated to UML. We mention here a few of its benefits: the ability to stack and align text in a visually pleasing way; a fine degree of control of how elements are laid out; the ability to group objects while having access to the properties of inner elements; a design pattern and syntactic sugar for writing modern-looking, reusable MetaPost code.

With this infrastructure in place, it should be possible to extend MetaUML until it offers complete UML 2.0 support.

References

Roegel, D. (2002). The METAOBJ tutorial and reference manual. Available from `www.loria.fr/ roegel/TeX/momanual.pdf`.

Knuth, D. E. (1986). *The TEXbook*. Addison-Wesley Publishing Company.

Lamport, L. (1994). *LaTEX a Document Preparation System*. Addison-Wesley Publishing Company, 2nd edition.

Gheorghies, O. (2005). MetaUML: Tutorial, Reference and Test Suite. Available from `http://metauml.sourceforge.net`.

Hobby, J. (1992) A User's Manual for MetaPost. Available from `http://www.tug.org/tutorials/mp/`.

Gjelstad, E. (2001). uml.sty 0.09.09. Available from `http://heim.ifi.uio.no/~ellefg/uml.sty/`.

Diamantini, M. (1998). Interface utilisateur du package pst-uml. Available from `http://perce.de/LaTeX/pst-uml/`.

Palmer, D. (1999). The umldoc UML Documentation Package. Available from `http://www.charvolant.org/~elements/`.

Object Management Group (2003). XML Metadata Interchange (XMI) Specification. Available from `http://www.omg.org/`.

Ovidiu Gheorghieş
Faculty of Computer Science
"Al. I. Cuza" University of Iaşi
Romania
`ogh@info.uaic.ro`