Visual Programming in Music

Gérard Assayag

Visual Programming in Music

Gérard Assayag

ICMC 95, Banff (Canada), 1995

Abstract:

PatchWork is a lisp-based visual programming environment for music. Propositions are made in this paper for the extension of PatchWork visual scheme to full object-oriented programming.

Introduction.

The PatchWork (Laurson 90, Malt 93) experience at IRCAM has proven the great interest shown by composers in visual programming. PatchWork has been distributed, through the IRCAM user's group, to over a hundred musicians and researchers in musicology. Many specialized libraries have been written at IRCAM and elsewhere adressing a wide variety of problems including new models for rhythm quantification (Agon 94), and musical constraints propagation schemes (Assayag 93). Composers that did not know a word in computer science have realized in PatchWork visual programs (patches) of an amazing complexity, due to PatchWork original mixture of visual functional programming and graphical data editing.

PatchWork runs on top of the Apple (now Digitool) Macintosh Common Lisp / CLOS (Common Lisp object System) environment (Steel 90). The visual scheme of PatchWork is purely functional, i.e. a graphical patch is an exact image of an embedded lisp function call where boxes are functions and links between boxes connect actual arguments to function parameters. Boxes may have a local state, which is in the simplest case the value computed by the function. This local state may be transformed by hand through a wide palette of editors (e.g. music notation editors).

We are now working on the extension of the PatchWork visual scheme to object oriented programming, i.e. we would like a graphical patch to be an image of a CLOS program. This paper will describe some propositions for visual extensions allowing the user to define classes, inheritance hierarchy, instances of classes, generic or polymorphic patches. In the last part of the paper, propositions are made for improving graphical control in higher level organization of musical substructures, using a new editor above the patch level.

The Icon interface.

A class is symbolized by an icon (fig. 1). Opening a class lets you visualize the class internal structure inside a new window. Icons in that window symbolize the classes of the instance variables (slots). New slots can be added by dragging and dropping icons from the current reservoir of classes. Icons can be opened in turn, giving access to the content of their associated class, in view-only mode.

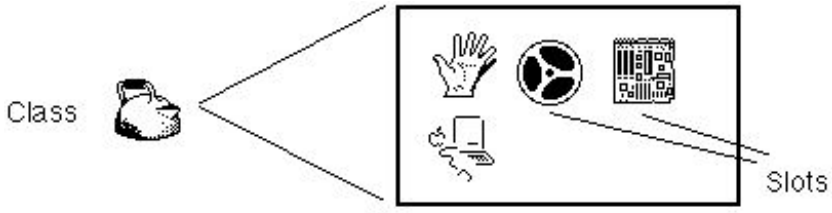

figure 1

Class hierarchies are designed by drawing links between icons (fig. 2). When opening a class icon that has been included in a hierarchy, the class window shows the parent classes instance variables in different layers, stacked from bottom to top in inheritance order. Only the bottom layer, i.e. slots belonging to the opened class, is editable. Instances of a class are derived graphically from the class icon, by a mouse operation (e.g. commad-drag). A new icon is then created for the instance. Opening an instance results in a new window whose internal structure is isomorphic to the window associated with the instance's class, except that icons here symbolize values or instances associated to slots rather than their class. Opening a slot's icon either gives access to it's substructure or to a specialized editor (e.g. music notation editor). Icons representing instances, just as constant boxes, can be provided as inputs to a PatchWork box.

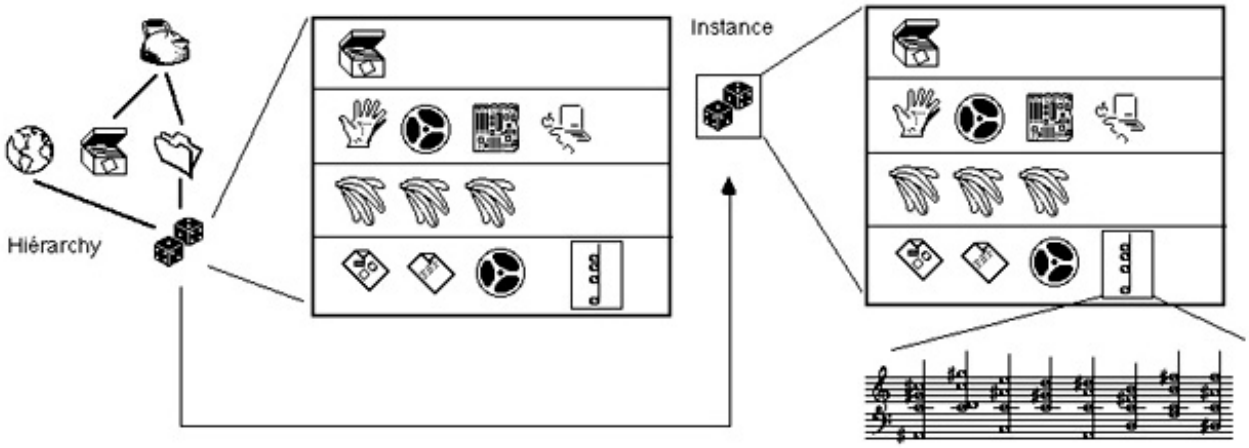

figure 2

The generic function interface.

Currently, PatchWork boxes are functions. There are two cases : either a lisp function (interpreted or compiled) is linked to the box, or a set of other interconnected boxes, implementing the body of a function, is associated with the box (in that case, it is called an abstraction). In order to be able to handle *generic* or *polymorphic* functions, i.e. functions that accept arguments of different classes then switch automatically to appropriate *methods*, the patch visual concept has to be extended. We shall call *generic boxes* boxes that dispatch on different methods depending on the type of their inputs, and *method boxes* boxes that materialize the methods themselves.

In the case where a generic lisp function and its methods are associated with a generic box, there is no particular interface problem. This is due to the fact that generic functions are smoothly integrated in the Common Lisp - CLOS environment and are considered externally as normal functions. Thus the usual mechanism will hold.

In the case where the generic box and its methods are programmed visually, we refer to the abstraction visual scheme. When a box is an abstraction, opening it results in the display of a patch window that contains a patch defining the abstraction. It is a normal patch except that it contains two kinds of special boxes : `absin` (abstraction in) and `absout` (abstraction out). `absin` boxes are the formal parameters of the abstraction. You will get as many rectangular inputs in the abstraction box than there are `absin` boxes in

the abstraction patch. `absout` is the output box : the value it computes is the value of the abstraction itself.

In the example in fig. 3, the "transform" box is a generic box. It performs a musical transformation with two inputs : a musical structure on the left side, and the functional transformation on the right side. Suppose "transform" is defined so that the musical structure can be specified either as a `chord` object (the output of a Patchwork box "chord") or a `BPF` object (the output of a BPF -- break point function -- box), and the transformation either as a `lisp-function` name ("MyFunction") or a `patch`. We introduce a supplementary level of box integration as can be seen in the second part of the picture. The boxes appearing in a new window when "transform" is opened are method boxes defining the different methods for transform. They are regular abstractions, which means they have associated patch windows containing a patch with `absin` and `absout` boxes. For each of these method boxes, one must specify the formal parameters classes. This is achieved by dragging class-icons into the window and connecting them to the inputs.

The top-left method in the sample window is defined to be called when the first argument to "transform" is an object of class `chord` and the second one an object of class `patch`. The top-right method has `chord` and `lisp-function` as input classes. The bottom-right is defined for the class `editor` and the class `everything` (the class T in CLOS). The bottom-left method is defined for `everything` on both sides. This last method is labeled `after`, which, in the CLOS terminology means it should be called after every other eligible method has been called. `before` and `around` methods could have been defined as well (Steele 90).

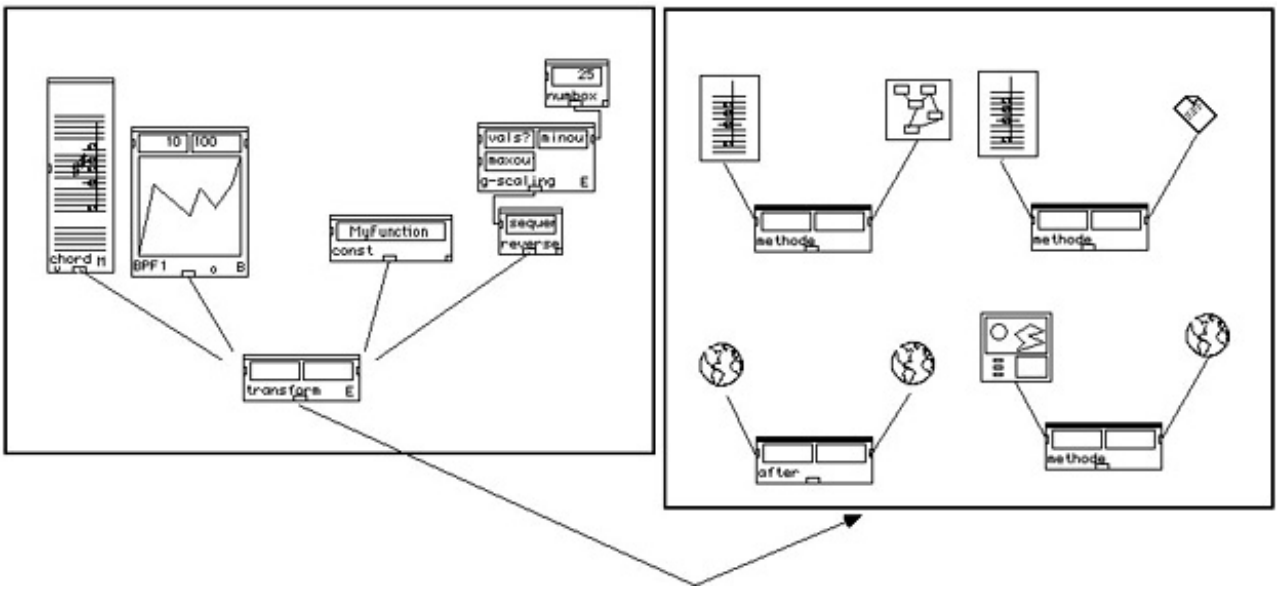

figure 3

Suppose "transform" is evaluated with the BPF box and "MyFunction" connected to it. As there is no specific method defined for the BPF class, and as both `chord` and `BPF` inherit from the `editor` super-class (fig. 4), the bottom-right method will be called, then the after method.

If a method wants to call the next available method in the hierarchy, it uses the call-next-m box (fig. 5). If this box appears inside the patch associated with the top-left method, it will act as an abstraction box for the patch associated with the bottom-right method, as `chord` inherits from `editor` and `patch` inherits from `everything`.

To end with, if a patch wants to access to a particular slot in an object that flows throught it, it may use a "slot" box. This box can be opened in order to show the internal structure of the associated class, if this may be determined at edit-time. Then the user selects a slot, or a set of slots, with the mouse (fig. 6).

As one can see, the patch visual concept can easily be extended to encompass many interesting features of CLOS.

figure 4

figure 5

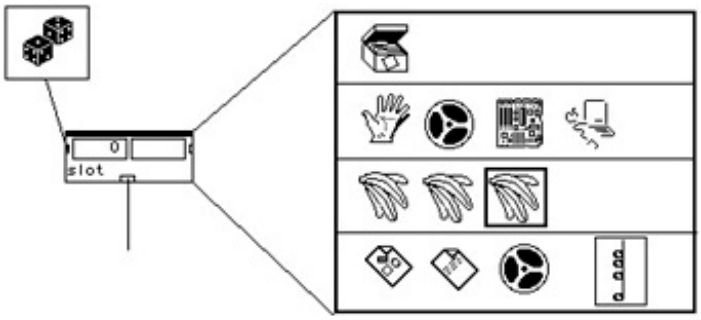

figure 6

Above the patches.

Using PatchWork, one can easily build a musical process modelled by a graphical patch, then connect the output to a musical editor module. Musical sub-structures are built that way. But it is still unpractical to combine sub-structures into higher level ones using concatenation and superposition . A new editor under design will let the musician create, display and manipulate graphical forms - currently simple rectangles - on a surface where time is one of the dimensions (fig. 7).

The idea is to associate a patch to each of these forms (1). The patch is responsible for computing the music sequence that will give a meaning to the form, through a special output box identical to the `absout` box in an abstraction. Thus, by controlling the spatial distribution of forms on the surface, the user controls concatenation and superposition of computed musical entities. As in Animal (Lindemann 90), the vertical and horizontal extents of the rectangles can be mapped to parameters into the associated patch. Obviously, the horizontal dimension will be interpreted as a time factor and eventually converted into a tempo in the case when it means something. Other graphical attributes like texture or curves drawn inside the rectangles will be mapped by the user to arbitrary control parameters in the patch. Selecting and double-clicking one or more forms will trigger the evaluation of the patches, combine the partial results, and open a music editor in order to display the sequence (fig. 8).

In order to set up relations between sub-stuctures, e.g. connect several of them into a single voice, links between forms have to be handled (2). These links are graphical objects associated to "glue" patches that tell how to perform the connexion. As the forms and their associated patches can be belong to different subclasses, glue patches have to be generic patches, with different methods for different classes of arguments. The user can then override the standard behaviour of these links by designing gaphically his own methods.

Another kind of link is a simple binary constraint that relate a variable in a patch with a variable in another patch, through a transformation function (3). This function is itself implemented as a patch. By using this feature, changing the value of a variable in a musical substructure may cause significant transformations in other parts of the score.

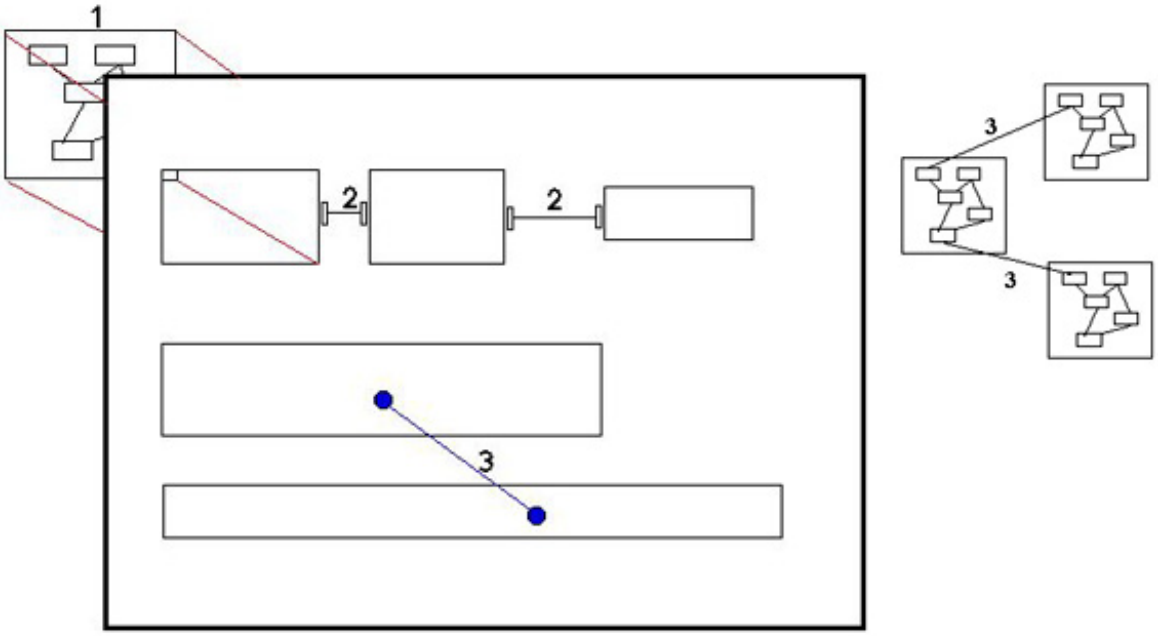

figure 7

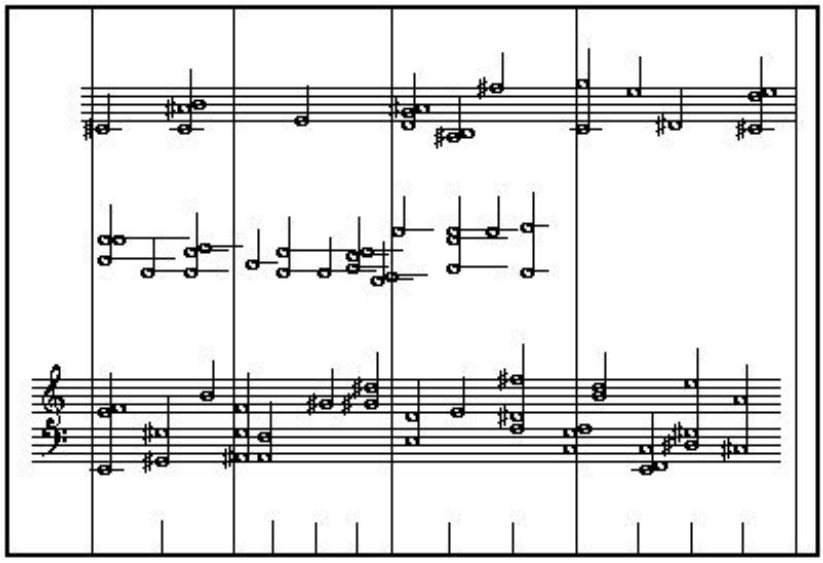

figure 8

References.

[Laurson 90] Laurson,M. Duthen, J. "A compositional environment based on Preform II, PatchWork and Esquisse." Proceedings of the ICMC 1990. Glasgow 1990.

[Lindemann 90] Lindemann, E. "Animal: A Rapid Prototyping Environment For Computer Music Systems". Proceedings of the ICMC 1990. Glasgow. 1990.

[Steele 1990] Steele G. "Common Lisp. The Language". Digital Press. USA. 1990.

[Malt 93] Malt, M. "The PatchWork Reference Manual". IRCAM, 1993.

[Assayag 93] Assayag G., Rueda C. "The Music Representation Project at IRCAM." Proceedings of the ICMC 93, Tokyo, 1993.

[Agon 94] Agon C., Assayag A., Fineberg J., Rueda C. "Kant: a Critique of Pure Quantification."

Proceedings of the ICMC 94, Aarhus, 1994.