High Level Musical Control of Sound Synthesis in OpenMusic

Carlos Agon, Marco Stroppa, G rard Assayag (IRCAM)
{agonc, stroppa, assayag}@ircam.fr

***Abstract*°*: new classes have been added to OpenMusic in order to handle the big amount of parameters involved in the control of synthesis.*

OpenMusic (Assayag & al, 1999) is a computer assisted composition software based on Common Lisp / CLOS (Steele 1990). It is is a complete visual programming interface to CL/CLOS where user operations are mainly based on the drag and drop scheme applied over *icons,* which stand for any OpenMusic-representable objects, and *containers*, which are editable panels giving access to the internal structure of objects. As CLOS is metaclass language, classes are a particular kind of objects, which are in that case instances of metaclasses. So classes may be handled visually as icons, and their structure built or edited through their graphical containers. The same stands for generic functions, which containers own a set of methods. OpenMusic being a musically-oriented environment, a lot of predefined classes and methods are made available to handle musical structures and allow their graphical edition (including, of course music notation).

After a lot of investigations in the field of symbolic computation, we have decided to explore the potential of the OpenMusic visual language as a control environment for sound synthesis. The idea was not to implement synthesis engines into OM, but to handle visually different high level control paradigms, then to generate the low level parameters towards whatever engine was available around. We have set up a collaboration with Marco Stroppa who has been involved for a long time in the control question and who had set up his own portable environment, called Chroma, in pure Common Lisp. The question adressed by Stroppa was in particular how to manage huge quantities of control data, bringing them back to a few, musically significant abstractions. He had designed a matricial representation, where rows and columns would set up vectors of values for identified synthesis parameters. The good thing was that values could be given as litterals as well as functions, thus providing the abstraction needed. Another advantage was that these functions could be easily connected to compositional processes, such as harmonic or rhythmical ones, independently from the constraints of synthesis. As OpenMusic provided already a powerful framework for composition, it seemed natural to implement this matrix abstraction in order to articulate symbolic composition and sound synthesis. It was also a challenge°: would the visual object oriented environment resist and prove ergonomy when confronted to the large sets of data involved by synthesis°?

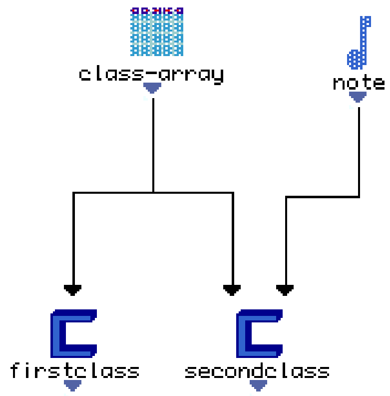

Figure 1a

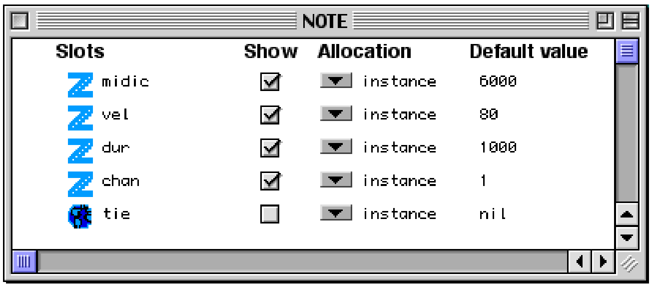

Figure 1b

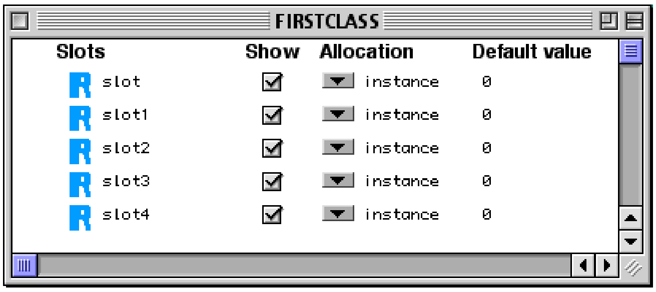

Figure 1c

A new abstract class named `class-array` has been added to the OM kernel. It is intended only to be subclassed. In the new subclasses, the slots will define parameters for the synthesis process. More precisely, each slot will be attached to a row in the control matrix. In picture 1, two subclasses have been defined. `Firstclass` defines a set of parameters (slot, slot4). `SecondClass` takes advantage from the underlying CLOS multiple inheritance scheme. Instead of defining new slots, we ll use the predefined slots of the `note` class (midic, vel, etc.) and these will constitute our synthesis parameters. Of course, the two approaches could be mixed, i.e., getting synthesis parameters both from existing musical classes and by adding new ones.

Once our classes have been defined, they can be used inside a patch in order to generate new instance objects. The way to do it in OM is simply to drag the class icon on a patch window.

At this point, OM creates a *factory*. A factory is a box which has inputs and outputs (figured as small round inlets and outlets) which are connected to the internal slots of the object to be created (fig. 2a). Upon evaluation, a new object is instantiated and its slots are initialized. A factory is generally associated with a graphical editor (Fig 2b), and with a miniview that shows the content of the editor directly on the patch. In figure 2a, only three among five parameters have been made visible as inlets. These parameters have been initialized in three different manners. From left to right : the number of columns (20), a litteral sequence of integer values (which will be cycled through untill the number of columns is exhausted), a sequence computed by a visual algorithm, and a visual-lambda expression (`myPatch`, opened in Fig. 2c). The latter, also called lambda-patch, is a function that is passed as a functional object rather than as a value resulting from computation. Thus, instead of computing a great number of values, we pass a function that will be used on the fly to provide these values when needed. This technique is called *lazy-evaluation*. In our example, myPatch computes the function y = cos(Pi/x). When the factory is evaluated, all the row-slots are initialized with the minimal information that will be necessary later for the full matrix to be computed. Note that the two last rows in the matrix miniview bear default values, as the inlets corresponding to their slots have not been used, and that the graphical editor can be used to adjust manually the values after computation.

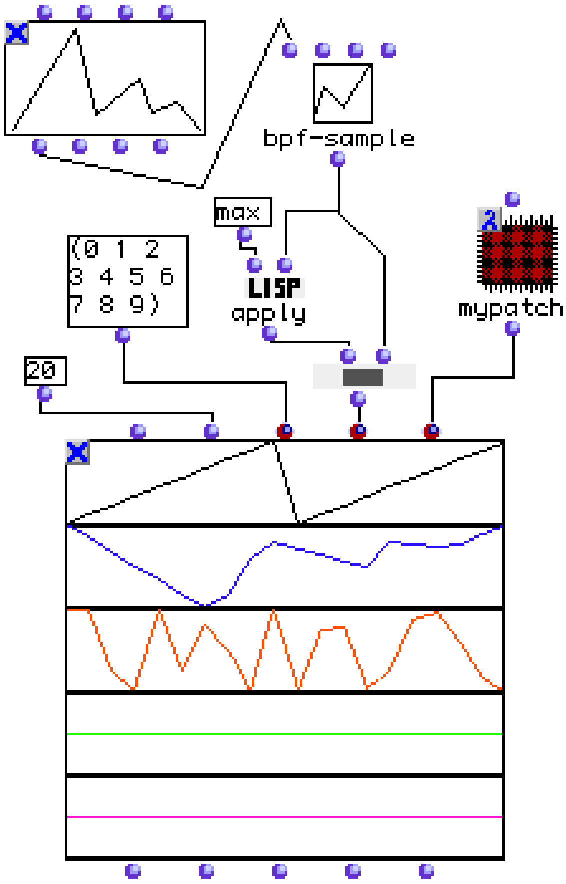

Figure 2a

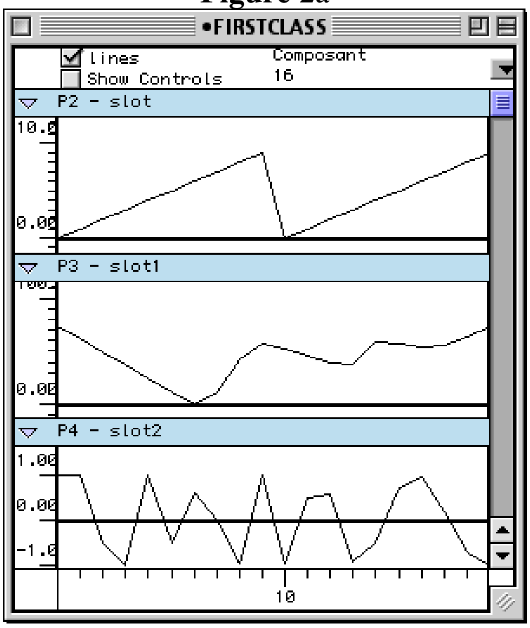

Figure 2b

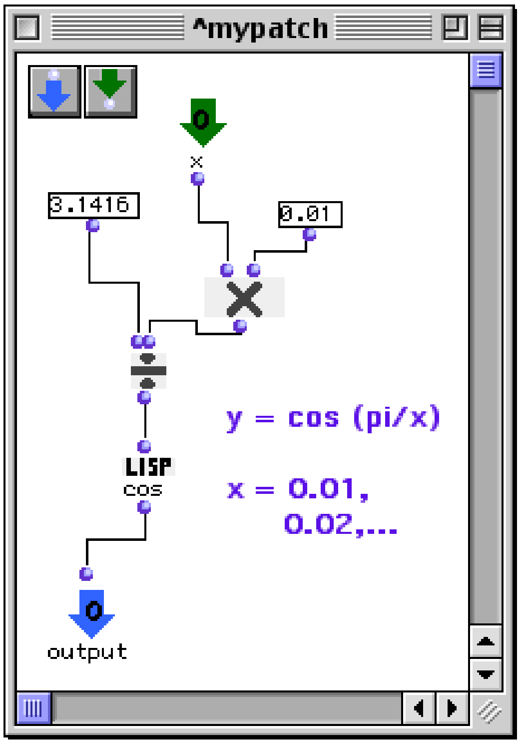

Figure 2c

In the example in fig. 2, the input types to the matrix are lists of numbers or functions that compute numbers. This is because the slots in class `FirstClass` have been defined as numbers (the R icon, fig. 1c). This will often be the case when the matrix is used for synthesis. But the slot type could be any type known to OM : e.g. `chord`, `voice`, `BPF`, or even a subclass of `class-array`. The only requirement is that slot initialization be made in a consistant way. So one can imagine initializing the cells of the matrix by providing a lambda-patch that will eventually fill them with other sub-matrices. The matrices defined by class-array are thus a very general tool of which only a small subset is now experimented in the domain of synthesis.

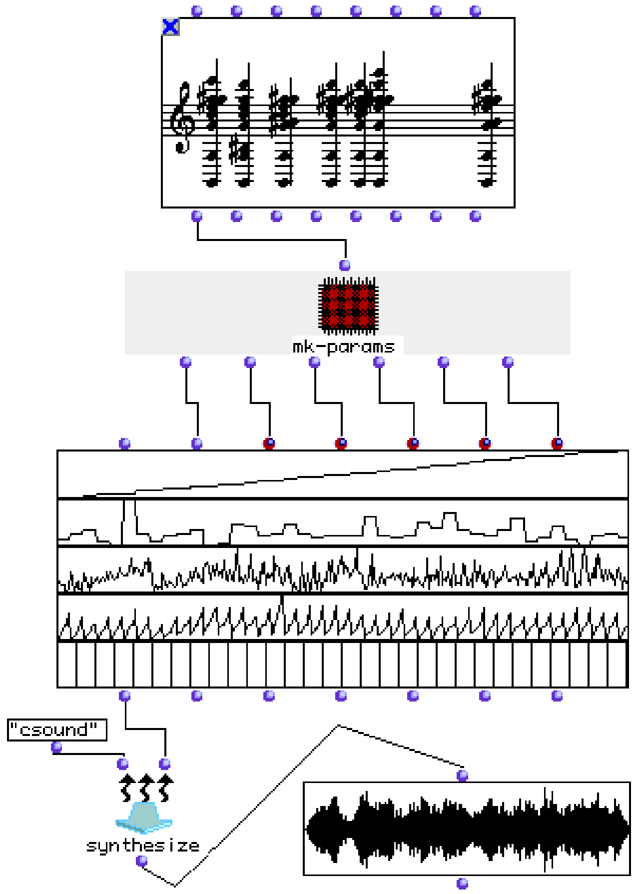

Figure 3

As it is defined, `class-array` matrices may be interpreted in many different ways as clusters of parameter values. The interpretation will obviously depend on the synthesis engine chosen and the on model it is based upon. We will now show an application to Csound (Boulanger 2000). We want to take a complex chord-sequence in OM as a harmonic model for additive synthesis. We have a very simple Csound instrument that knows how to synthesize a simple partial. A parser in OM takes the .orc file, recognizes the parameters (from p1 to p6) and generates automatically a subclass of `class-array` with as many slots as needed (class `SinusInst`). In figure 3, a `SinusInst` factory is shown. All its onset, duration, frequency, amplitude, amplitude envelope table, are computed from the `chord-sequence` factory above, by the subpatch `mk-params`, which we do not detail. Every column in this factory is an internal representation for a command line in the Csound score file, of the form e.g. « `i1 0.0 2.15 0.496063 86.603877 2000` ». Every line is a different parameter defined by a particular slot and its associated inlet. Note that the choice of this interpretation of the matrix structure is arbitrary, and is made here because it is the most practical when thinking about the Csound style of control. When applied to another engine, another interpretation will eventually be necessary. The interpretation of the matrix data is concentrated into the method `synthesize`. This method will dispatch automatically to a score file generation function (in the case of Csound) or even to a function that will talk in real time through a communication channel with such real-time engines as Max or jMax. It is the `synthesize` method that decides the interpretation of the matrix, depending on the target synthesizer. Currently, a `synthesize` method is available for°: Csound, Chant, and Modalys. Another useful application under way is a method that generates SDIF files°: OM will then be compatible with all the engines that recognize this file format.

REFERENCES

Assayag, Rueda, Laurson, Agon, Delerue, 1999, ˙°Computer Assisted Composition at Ircam : PatchWork & OpenMusic°¨, Computer Music Journal 23:3.

Boulanger R. (Ed.), 2000, ˙°The Csound Book-Perspectives in Software Synthesis, Sound Design, Signal Processing,and Programming°¨, MIT Press.

Schwarz D., Rodet X. et al, 2000, ˙°Extensions and Applications of the SDIF Sound Description Interchange Format°¨ Proc. ICMC 2000, Berlin.

Steele, G.L. ˙°Common Lisp The Language¨ 2nd Edition. Digital Press. 1990.