OBJECTS, TIME AND CONSTRAINTS IN OPENMUSIC

Carlos Agon, Gérard Assayag, Olivier Delerue, Camilo Rueda.
{agonc,assayag,delerue,crueda}@ircam.fr
IRCAM - 1 Pl. Stravinsky. F-75004 Paris, France.

1. Abstract

OpenMusic is a object oriented visual programming environment for music composers. It is based on Macintosh CommonLisp and Common Lisp Object System. It shows several original features, such as reflexivity, metaprogrammation capacities, handling of the duality between musical time and computational time, and provides with a framework of predefined musical objects for handling sound, midi and musical notation. Common Music Notation in OpenMusic is an extension of CMN, a public domain notation package by Bill Schottstaedt [SCO98].

2. Objects

2.1. Underlying Object Model.

Broadly speaking, basic calculus for object-oriented programming inherited the approach imposed by the precursory language Simula and its followers. The tentatives to formalize this family of object models used the concepts of parametric polymorphism or true polymorphism [CaWe85]. More recently, languages based on multiple-dispatching (methods that dispatch on a product of types rather than a single type) such as CLOS [Steel90] were also formalized [Cast98], using concepts of overloading or ad-hoc polymorphism. As OpenMusic, based on CLOS, may very well be formally described in this way, we will give here a brief summary of the λ&–calculus used by [Cast98]. We restrain to the extension of λ–calculus by the concept of generic functions which is at the base of λ&–calculus. A generic function is simply a collection of simple functions (λ–abstractions), which we will call methods. We must also distinguish the simple application indicated by «.» from the application of generic functions indicated by the operator . Thus a generic function with n methods can be specified by the expression: $(\varepsilon \,\&\, M_1 \,\&\, M_2 \,\&\, ... \,\&\, M_n)$. The type of a generic function made up with methods M_i of the type $U_i \to V_i$, is defined by: $\{U_1 \to V_1, U_2 \to V_2, ..., U_n \to V_n\}$. In λ&–calculus, however not any set of method types can be seen as the type for a generic function. A set of methods types $\{U_i \to V_i\}_{i \in I}$ is a generic function type iff for all i,j ∈ I the two following conditions are satisfied:

$$U_i \le U_j \quad \Rightarrow \quad V_i \le V_j$$

$$U \text{ is maximum in } LB(U_i, U_j) \quad \Rightarrow \quad \exists! h \in I \quad , \quad U_h = U$$

where LB(U, V) indicates the set of common lower boundaries of types U and V. This enforces consistency in the case of multiple inheritance. The application of a generic function G to an argument N of type U consists in the selection of a method M_j among the methods of G, then the application of M_j to N.

$$(\varepsilon \,\&\, M_1 \,\&\, M_2 \,\&\, ... \,\&\, M_n) \bullet N \quad \triangleright \quad M_j.N$$

Note that U may not be contained in the set U_i of input types to the generic function. In such case we select the method M_j satisfying : $U_j = \min_{i=1...n}\{U_i | U \le U_i\}$. The $\le$ ordering is defined by the class precedence list CPL. The CPL for class C is a topological order of the superclasses of C. An object in the λ&–calculus is a simple register $\langle l_1 = T_1, ..., l_n = T_n \rangle$. A register can be seen as a set of labelled fields l=T where l is called label and T is called value. There is a reduction rule for the field selection : $\langle l_1 = T_1, ..., l_n = T_n \rangle.l_i \quad \triangleright \quad T_i$. We add also the context rules for general expressions in λ&–calculus: $E \triangleright E' \Rightarrow (E.l \triangleright E'.l) \wedge (\langle ...l = E... \rangle \triangleright \langle ...l = E' ... \rangle)$. If objects are seen as registers, classes will then be seen as generators of registers. We assume that there is a generic function «new» and that each class defines a method for this generic function. We also assume in what follows that the type of an object is its class. This is the rule for subtyping in the case of generic functions:

$$\frac{\forall i \in I, \exists j \in J, S_j \to T_j \le U_i \to V_i}{\{S_j \to T_j\}_{j \in J} \le \{U_i \to V_i\}_{i \in I}}$$

Thus, the inheritance mechanism for the λ&–calculus is defined by the subtyping and the mechanism of method selection. If we call a generic function of type $\{C_i \to T_i\}_{i \in I}$ with an instance of the class C, then the method

$\min_{i=1...n}\{C_i | C \le C_i\}$ will be called. If C_i is C, then the object uses the method defined for its class, on the other hand if $C_i > C$, then the object uses a method that its class inherited.

In the approach described here, objects are passed as arguments to generic functions. In the classical approach, messages were rather passed to objects. Significant feature of the λ&−calculus are :

- it allows the multiple dispatch (i.e. capacity to select a method by taking into account other arguments than the first one).
- it makes it possible to add methods to an existing class without modifying the class and its instances. This is a critical issue in OpenMusic which is a fully dynamic environment.
- generic functions are first-order citizens ; for example we can write the following expression: $\lambda m^{\{C \to C\}}.\lambda x^{C}.(m \bullet (m \bullet x))$

2.2. A Visual Object Model.

OpenMusic is a multiple-dispatch, generic-function based visual object programming language. A formal description of the lexicon, syntax and semantics of OpenMusic and its precise relation to λ&−calculus will be found in [AgAu98]. In this section we show only a pragmatic description of concepts taking part in our object model.

A program (patch) in OpenMusic is a graphical layout on the screen, made of composed frames or simple frames. Composed frames contain simple frames or are empty. The objects belonging to our calculus, (i.e. classes, methods, slots, generic-functions, instances), and its rules (i.e. multiple-inheritance) are visually represented as composed frames or simple frames. Several different frames (i.e. different representations or point of views) may be produced for the same object. The simple frames representing an object are called object views. They generally appear as icons. The composed frames representing an object are called object containers. They generally provide a graphical editor for the object. The container for a class is an ordered collection of simple frames representing slots. Slots have information about their name, their type, a default value and a flag that indicates if the slot is public or private. Figure 1 shows the structure of the class « chord », which containts 5 slots belonging to type list.

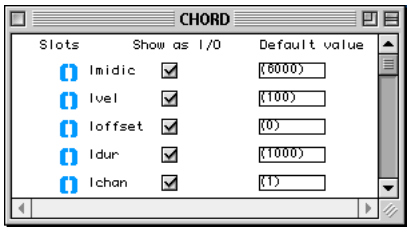

Figure 1. A chord class.

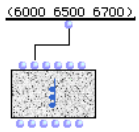

Figure 2. A chord factory.

Users can create instances of the chord class with the aid of a particular box called a factory (fig. 2). A chord factory contains a number of inputs corresponding to the public slots of the class plus one special input on the left. There are as many outputs as inputs. When evaluating (i.e. option-clicking) an output a new instance is created. The values returned by outlets are, from left to right, the instance itself, then the current value of its public slots. An instance can be visualized graphically as an icon, allowing its use within a patch (fig. 3). There exist different types of containers for the created instance. Fig. 3 shows a common music notation editor and a structure editor for the same instance of class chord. The last editor, although very primitive, is also very general in the sense that any kind of instance can be viewed and edited that way. The editor shows the layers of inheritance (a chord class inherits from a more general superposition class). Slots can be edited here by directly typing in values or dragging objects.

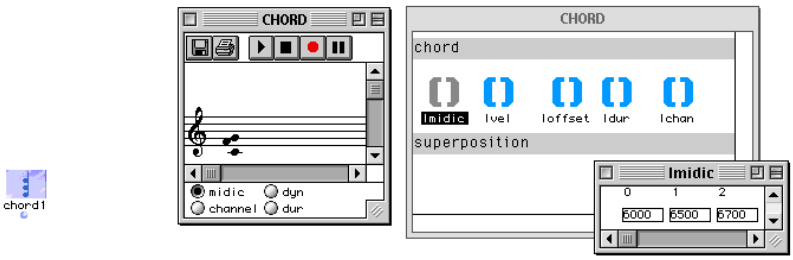

Figure 3

The container of a generic function is visualized by a set of simple frames representing methods (fig. 4). A simple frame for a method contains the name of the generic function that owns the method, an icon for that function and a list of type specializers (small icons on top) for each parameter of the method. In our example, the generic binary function « transpose » is defined for the couples of types (voice , integer), (note, integer) and (measure, integer). Users may, with the aid of a method editor, create new methods or modify already existing ones. Fig. 4 shows the graphical definition of the methode transpose for a chord and an integer. The icon representing the first input to the method has small outlets that represent the public slots of class chord. The box « slots » is reader-writer box automatically defined for each class in the system. In the same way, user-redefinable initialisation methods are automatically generated.

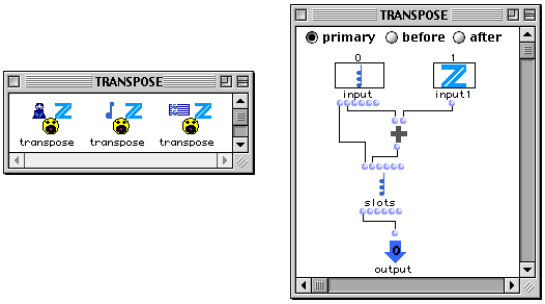

Figure 4

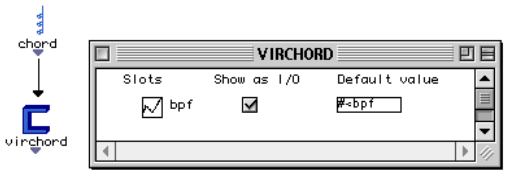

Figure 5

In much the same way, users can create and redefine their own classes. In the following example (fig. 5) we create a new class called virchord which inherits from the class chord. In the composed frame representing inheritance, a new class icon is added, with an arrow that symbolizes the inheritance path. When opening the virchord icon, the user has access to the class editor, where he may add and edit new slots. The slot class is specified by dragging a class icon into the frame.

2.3. Implementation.

Rather than a simple graphical interface to CLOS, *OpenMusic* (OM) may be regarded as an extension of CLOS with metaprogramming techniques. This involves basically the possibility of subclassing metaclasses (such as standard-class, the class whose instances are themselves classes) and then defining new methods for these subclasses. Fig. 6 shows the hierarchy of OM classes. CLOS classes are in bold framed rectangles.

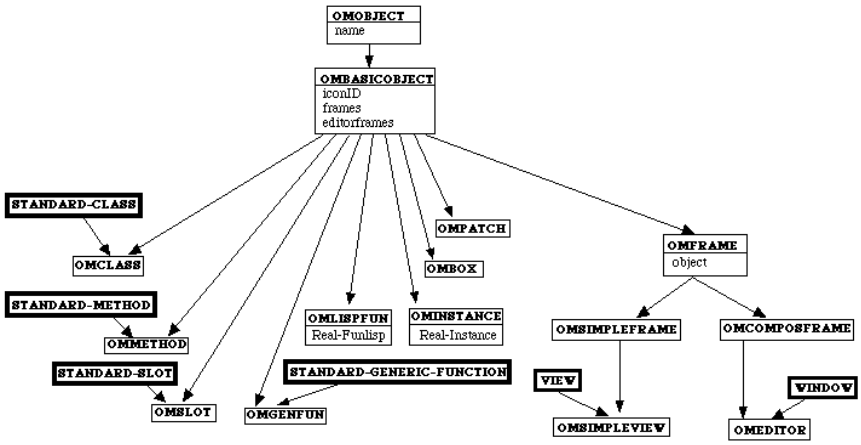

Figure 6

Almost all CLOS meta-classes have been orthogonally integrated in the visual formalism. The left side of the tree in fig. 6 defines OM metaclasses whereas the right side (rooted at OMFrame) implements the visual paradigm. Every OM object has a unique name and it points to a list of frames of class OMsimpleView as well as a list of editors of class OMEditor. Note that class OMFrame inherits from OMBasicObject. This means that classes that describe the visual behaviour of OM are themselves ordinary OM objects, and may be manipulated as such. The graphical part of this class design is very general and flexible and might be easily implemented in another environment.

As for the dynamic part of this model, we will consider that an object is composed by a set of elements together with a relation over them. A class object, for example, is composed by an ordered list of slots, a generic function consists of a set of methods, a patch contains a list of boxes, an editor a list of frames, etc. A protocol of generic functions that are applicable to all instances of OMBasicObject has been defined in order to control their behaviour. Examples of such functions are :

- get-element which returns for any OMBasicObject the list of its elements.
- get-simple-view and get-editor return the two possible graphical representations of an object.
- open-editor-object calls the function get-simple-view for each element of the object. An OMEditor is then constructed with all these simple views as elements.
- add-element and remove-element allow editing of any basic object.

These last functions are mainly called by the drag&drop mechanism, which is a central issue in OM user interface. It is defined as an action between a dragged OMsimpleView and a target OMEditor. A set of allow-drop predicates determine whether the source object slot can be added to the target object slot. The drop operation is performed only when those predicates hold. This results in the invocation of the add-element function with arguments dragged and target. First, add-element makes the drop operation visually explicit. Then it delegates the message add-element to the objects pointed at by the « object » slots in the dragged and target frames, so as to launch the edition of the relevant OMBasicObject. Other OM operations such as move, select and eval follow a similar mechanism. This set of methods defines the dynamic part of the language whereas the class tree defines the static part.

2.4. Visual meta-programming

In the same way we extended CLOS by using the technique of metaprogramming, the user can make extensions to the OpenMusic language. The principal tools for the metaprogrammer are : subclassing inside the static class definition part and redefining functions in the dynamic protocol part. Fig. 7 shows the hierarchical tree of metaclasses and the generic fonctions defined for each class. The reader can be surprised by the simplicity of the protocol. In general, a protocol too much specified is not very modular, thus changes must be done in several places, which makes the modifications not very reliable ; on the other hand poor specification of the protocol produces a difficulty to find the place for including modifications. This is the best compromise we found.

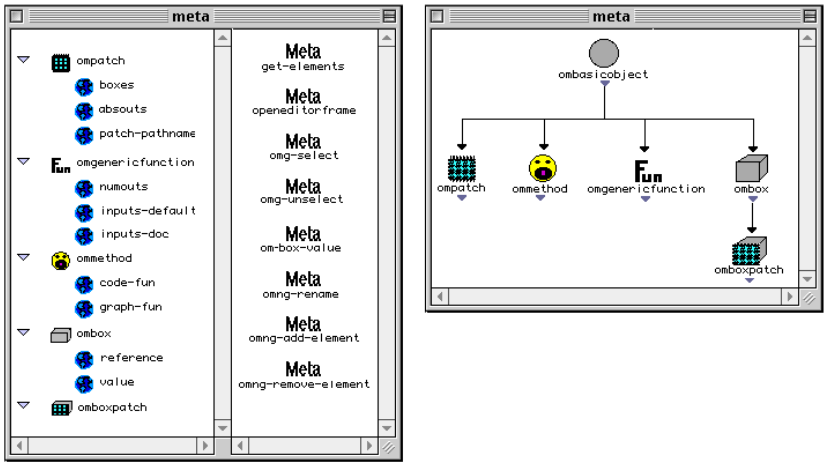

Figure 7

The following example (Fig. 8) shows the redefinition of the generic protocol function OM-box-value which is called whenever the output of a box is evaluated. Graphical redefinition of its three methods changes the behaviour of the language by introducing a visual trace of every evaluation. First method, defined with qualifier « before », selects any currently evaluated box before its execution. The second method, with qualifier « after », unselects the box after its evaluation. Finally the method defined for the classe OMBoxPatch opens the patch editor window for any currently evaluated subpatch. It then calls the function OM-box-value on the object (arrow icon) that represents the output of the patch. This output object will be highlighted (due to the « before » method), then propagate the om-box-value message to connected boxes, causing them to highlight in turn, and finally unhighlight because of the « after » method.

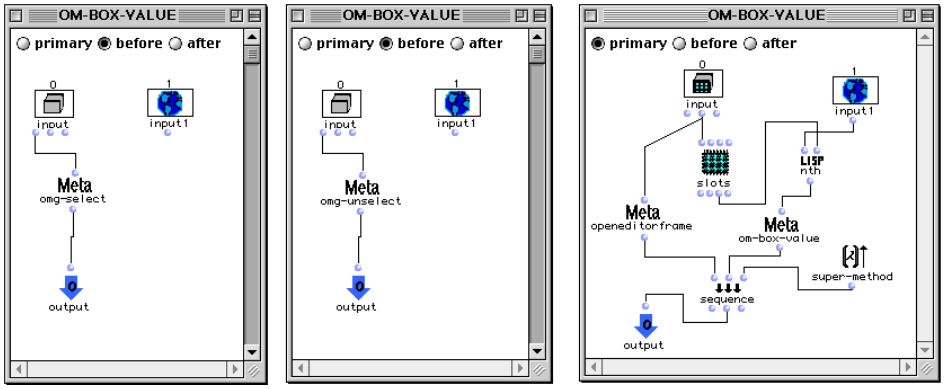

Figure 8

2.5. Musical Object Framework.

A summary of the predefined musical classes currently available in OM is given in fig. 9.

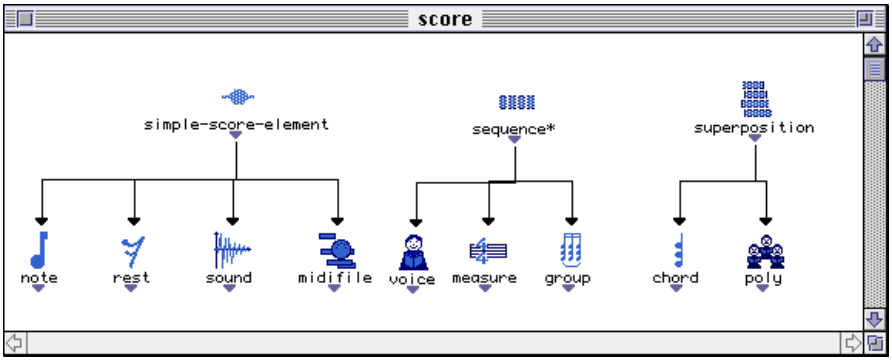

Figure 9.

3. Time

3.1. Metric Time

A set of elementary rhythmical operators has been defined and implemented in Open Music. These work on musical structures such as voices and take advantage of this hierarchical representation by preserving the embedment information of the initial data into their results. We describe here two examples of such rhythmical operators.

3.1.1. Fusion

First is the « fusion » of musical structures : this operation consist in merging together two different voices. The result is a voice that contains simultaneously the notes in both the initial structures.

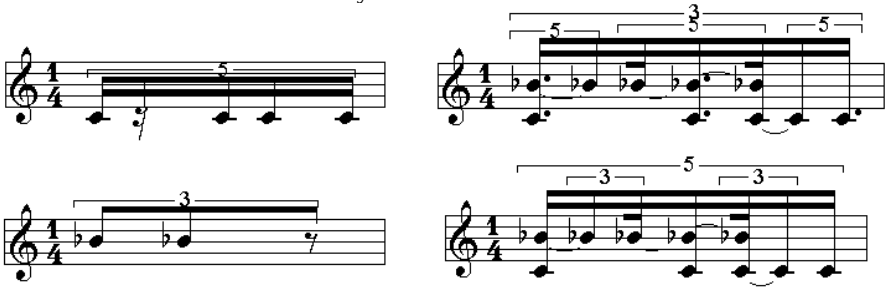

Figure 10: example of fusion of musical structures

In Figure 10 we describe on the left the initial voices that were chosen and, on the right, two different results of this operation according to the choice of which of the initial voice will include the other. Experience shows that tuplets embedments are simpler when the higher irregular subdivision includes the other.

3.1.2. Masking operation

Another example of rhythmical operation is the masking of musical structures. This operation is very close to the previous one, in the sense that their algorithms are very similar. It operates on two voices and performs a masking effect on one of them (data voice), using the other one as a mask (masking voice). As a result, the data voice is muted during the masking voice's activity or inactivity. In other words, notes in the masking voice will be propagated as rests in the data voice if the masking parameter is set to « activity ». The « inactivity » alternative propagates rests of the masking voice into the data voice.

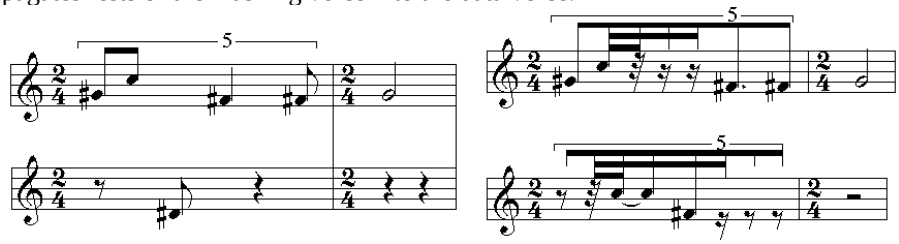

Figure 11: example of masking of musical structures

On Figure 11, the left system shows the initial voices chosen for the masking operation : the upper staff is used as data whereas the lower one is used as a mask. On the right, we describe the two different results of this operation depending on whether the masking parameter is set to « activity » or « inactivity ». The fusion of these two resulting voices brings back to the initial data voice.

3.1.3. Regularisation

Eventually, another complementary family of operators is being developed : these aim at building a relevant musical structure from objects that either don't have yet any structure or are too complex from the musical notation point of view. Indeed, Open Music allows for instance to define musical structures starting from numeric values obtained as the result of an algorithmic calculation and interpreted as a list of durations. These values may not however be directly usable for musical notation and need to go through a process that will ensure the compatibility of the resulting structure with the musical notation system.
For instance, the « regularisation » operator will perform such a task by detecting irrelevant irregular subdivisions within a musical structure and by suggesting a compromise between adding embedding levels and quantification.

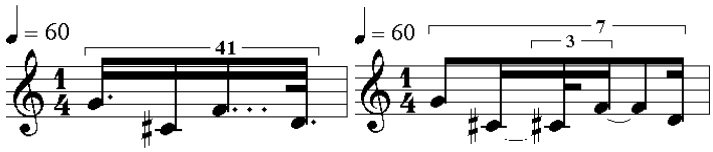

Figure 12 : example of the regularisation operation.

Figure 12 shows an example of the « regularisation » operator : we define here a musical structure starting from the following list of duration (12 8 15 6) that we wish to represent within the duration of a quarter note. On the left we represent this structure as is, i.e. without any transformation. On the right, this structure has been transformed by this operation by a quantification factor of 42/41 of a quarter note and adding an embedding level. The four notes still have the duration of a quarter note, but the *monnayage* is now (12 8 16 6) which means the F natural has been slightly extended with respect to the other notes' duration.

3.2. Maquettes : the duality of musical and computation time.

The maquette is an original concept in OM by which we adress the problem of combining design of high level hierarchical musical structures, arrangement of musical material in time, and specification of musical algorithms. Simple patches may adress partially each of these problems but obviously fail to provide with a fully integrated solution. Just as patches, maquettes may appear as an icon abstraction or be opened in a maquette editor, which is basically a 2-dimensional surface with time flowing along the x-axis. On this surface, temporal-boxes are laid down. Fig. 13 hsows three different representations for a maquette.

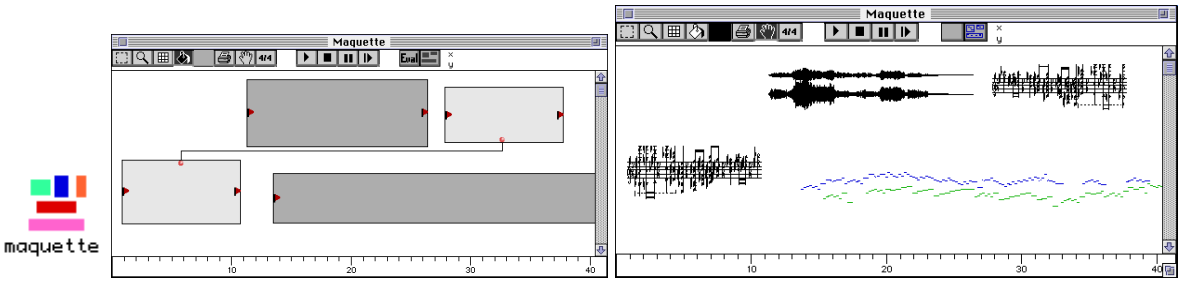

Figure 13

Just as classical boxes in a patch window, temporal-boxes have a reference (the OM object they stand for) and a value (the result of the last evaluation). In addition, there are 2 slots, offset and extend, that specify the position and the time-span. Fig. 14 shows the class tree. There are 3 types of references for a temporal-box :

- a temporal-object (an object that can answer to questions like start-time? and duration?). Musical objects like chord, voice, etc. are obviously temporal objects.
- a patch. A patch is not a temporal-object, but it may of course compute a temporal-object and deliver it as a result. Patches that are a reference for temporal-boxes are special patches called temporal patches. An editor for a temporal patch is shown in fig. 15 Any algorithm may be described here in the usual way (including call to subpatches). By connecting the output of any box to the « tempout » special output, we state that the output of this box do construct the temporal object we want to lay down at this place on the maquette.
- a maquette. Thus maquettes can be embedded into other maquettes.

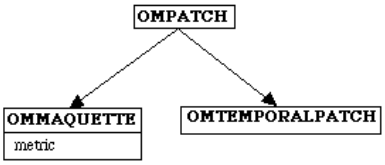

Figure 14

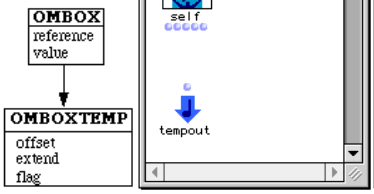

Figure 15

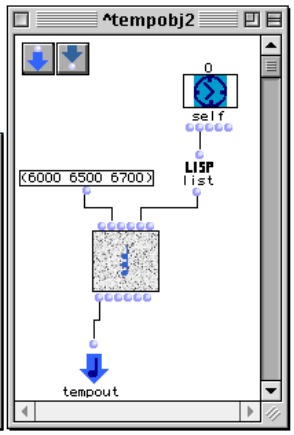

Figure 16

A protocol of reader-writer generic functions is defined for handling algorithmically temporal-boxes in a maquette : objdur (get/set the duration), starttime , mymaquette (get the owning maquette) , put-in-maq (insert a box in a maquette).

The special box « self » (fig. 17) stands for the temporal-box itself. The public slots available as outlets are offset, extend, reference, and flag (not discussed here). In fig. 17 we see a temporal-patch associated to a temporal-box, where the temporal-object is just a chord whose pitches are input as a list, and whose duration is directly derived from the offset slot from the self object. In that case, every time the temporal-box is moved around in time by the user, its duration will change accordingly.

In Figure 17 we add a supplementary output to our temporal-patch and connect the pitch-list slot of the chord object to it. This output will appear as a small outlet on the temporal-box inside the maquette editor. In another temporal-patch, we add an input. We have now 2 temporal-boxes in the maquette, that we can interconnect. As can be seen, the pitch-list from the chord is reversed , transposed by 5 semitones, and input to a sequence factory, resulting in a melodic motive. We have now a maquette with a chord whose duration depends on its horizontal placement, and a motive that depends on the content of the chord.

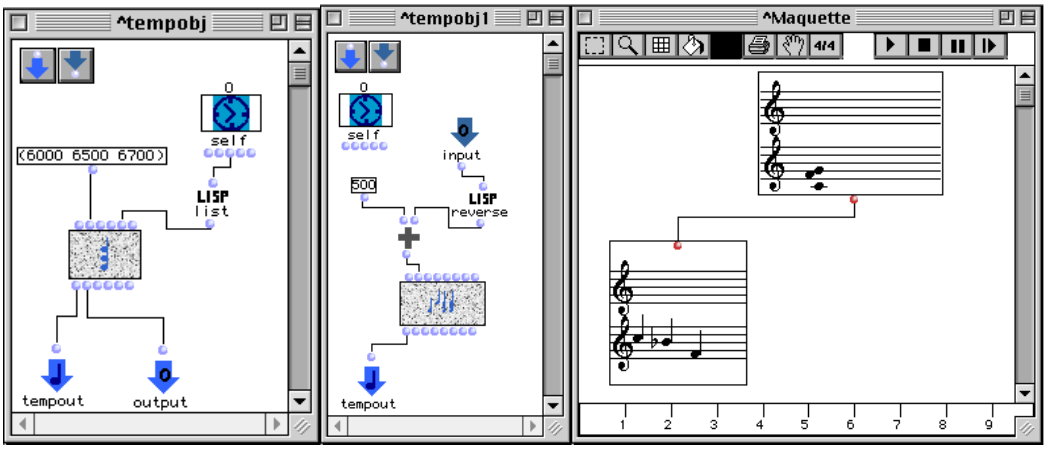

Figure 17

Pushing this idea further, Figure 18 shows a sketch for a piano piece realized by composer Mikhail Malt. For the sake of simplicity, connections have been hidden, but the temporal-boxes are highly interconnected. Using a facility provided along with maquettes bitmap images have been dragged over the boxes in order to suggest graphically musical processes.Vertical triangles are chords whose contents are propagated to other boxes through connexions. Other boxes contain musical objects defined by new classes like chord-ostinato, trill with amplitude modulation etc. By changing the temporal-patches that define chords, all the pitch material is changed accordingly in the piece. By moving around the boxes, or stretching/compressing them, all the time organization is changed while keeping the logic that links the different pieces of material.
Finally, by using the temporal-boxes protocol mentioned above, temporal-patches may generate dynamically (e.g. by cloning themselves), when evaluated, other temporal-boxes, or manipulate existing one (e.g. moving or scaling in time). In that case, the user would proceed in two steps : first, design statically a maquette, with interconnected boxes containing musical material or patches for constructing it ; second, evaluate the maquette , seeing new boxes appear, other move or even disappear.

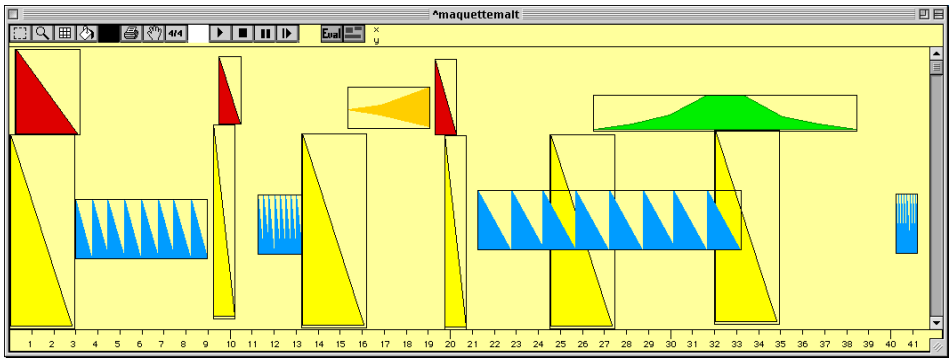

Figure 18

There is here the new concept of a score where the static description of musical structure and musical time, and the definition of dynamic computational processes seamlessly coexist. The user may go back and forth between these two metaphors by choosing to see the maquette as a score (in traditional or graphical notation) or as a set of interconnected processes. As external objects (e.g. MidiFiles, SoundFiles) may as well be imported by a simple drag, maquettes may provide with an original environment for music creation.
Thus the title of this section, inspired by [Pratt92].

4. Constraints

4.1. Underlying Constraint Model

There is a built-in constraint solver in OM, called Csolver. A particular musical interface layer, called Situation and expertised by composer Antoine Bonnet, allows for specific muscial constraints solving. CSolver allows the construction of objects out of two notions: point and distance. An object contains one or more points, spaced according to distances which are evaluated in a given unit of measure and with respect to a user supplied distance function. Distances can be internal or external depending on (respectively) whether they involve points contained

in the same object or in several different objects. Intervals in a chord, for example, are internal distances whereas melodic intervals are external. The unit of measure can in this case be the semitone, the eighth tone or any other. By default, points are considered to have integer coordinates in a standard cartesian space. The sequence [60,64,67], for example, could represent an object (chord) consisting of three points (notes, pitch expressed in Midics) separated by distances of 4 and 3 semitones. The objects could also be rhythmic plans concerning a set of articulation points spaced according to a given set of temporal distances. An eighth note, quintuplet or other could be the unit of measure in this case. The sequence [31/4,65/8,67/8], for example, might represent the articulation points of a voice object. The unit of measure is 1/8 (an eighth note). The first articulation point is located at position 31/4. 3/8 and 1/4 are the distances separating consecutive points in this voice.

The specification of a problem for CSolver consists in supplying the number of objects wanted, the space of possibilities (region) for points, the number of distances in each object and the set of possible values for these distances. Built-in constraints establish the allowed configurations of each object and also that of object combinations. Any point or distance of any object or set of objects can be related by a constraint. Built in constraints conveniently define : general profiles that should follow the objects ; patterns that a given set of distances should match ; predetermined points that should belong to every solution ; equality or difference of points or distances (or simultaneity, in the case of rhythms), etc.

CSolver is a finite domains system. Each computed object is by default represented with two domain variables. The first variable defines the position of the first point of the object and the second variable defines the sequence of distances separating each consecutive point in the object. The domain of the first variable is any finite set of numbers. The domain of the second is a finite set of sequences of numbers. The latter domain is usually large. CSolver allows it to be structured in a tree hierarchy. A subset of the sequences sharing a given property can be collected into a subtree in this hierarchy. At the level of the root of this subtree, the whole subsequence is represented by a domain of just one value: the shared property. A collection of (user supplied) functions computes properties to be used to abstract the sequences domain at a specified level. By default, CSolver uses two levels, with the sum of the object's distances as the abstracting property for the highest level. This is very convenient for harmonic problems, where the upper and lower voices are usually more constrained than the others.

The notion of "distance" in CSolver is not fixed. The composer can define her/his own by supplying the appropriate functions (normal, inverse) and neutral element. In musical applications this option can be very important. For example, some composers conceive harmonic material as aggregates of frequency partials related in precisely defined ways. Multiplicative distances are more relevant in this case.

The search engine of CSolver uses first-found forward checking [RuVa97], a lazy-evaluation version [DeMe94] of forward checking extended to hierarchical domains. Each domain level keeps track of the position of the current consistent value at that level. These are values known to satisfy all constraints referring to that level. A judicious choice of data structures allows CSolver to efficiently update these current positions as new constraints are checked or when backtracking is needed.

No variable reordering is used (although included as an option) since constraints for musical problems generally apply within short subsequences, with little dependencies across subsequences. All domain values are randomly permuted prior to exploration. The reason for this is that the musician is in most cases interested in obtaining few but widely different solutions to a given problem. Due to domain permutations, any new execution of the same problem is likely to give a solution with different values for many of the variables.

Constraints can relate any level of the domain of one variable to any level of the domain of any others. Arc-consistency [Mack77], via AC-7+ [VaRu96] an enhancement of the algorithm in [FBRe94] can also be performed for binary constraints over upper domain levels.

Figure 19 shows an automatic harmonization of the beginning of Debussy’s Syrinx (dragged in as a midifile). There is a profile constraint for the bass that forces it to monotonically decrease, vertical interval constraints that allow thirds between chord 0 and 7, then add augmented fourth. Other constraints on the density and ambitus profiles are not shown here.

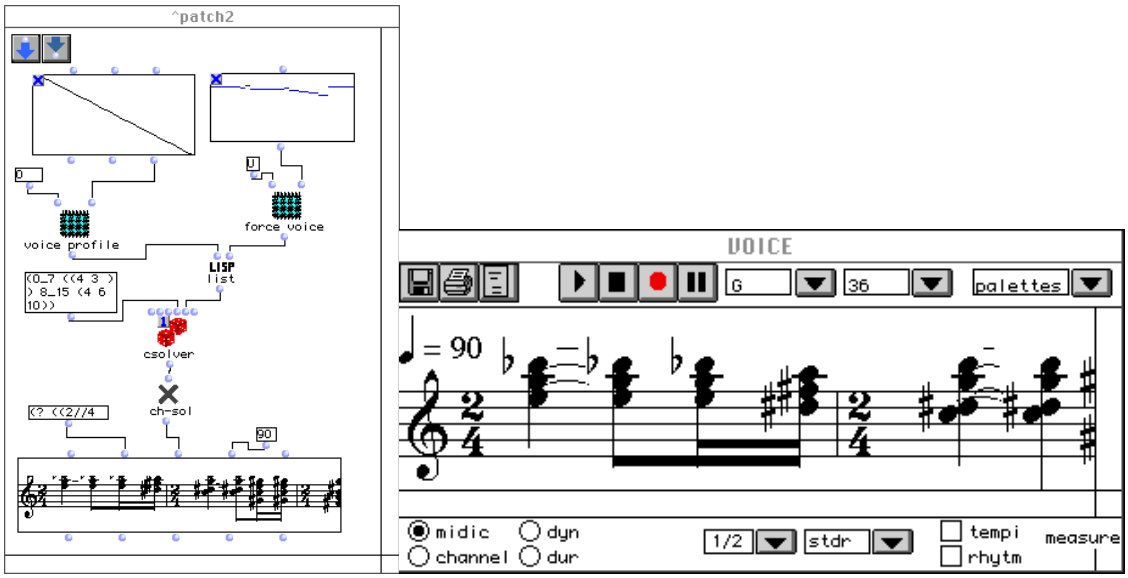

Figure 19.

5. *Future Works*

Future direction for OpenMusic development are : visual abstractions for sound synthesis, improving music notation display in maquettes, refining the protocols for metaprogramming and dynamic construction of the maquettes.

6. *Aknowledgements*

We wish to thank Bill Schottstaedt for his kind help on CMN, and Mikhail Malt for his support and brillant early experimentation with OpenMusic.

6. *References*

[Ago98] A. Agon. An environment for computer assisted composition. PhD Thesis. IRCAM-Paris VI. Paris. To come.

[Cast98] Castagna Giuseppe. Foundations of Object-oriented programming. ETAPS, Lisbone, 1998.

[CaWe85] Luca Cardelli, Peter Wegner. On Understanding types, data Abstraction and polymorphism. Computing Surveys vol. 17 No 4, 1985.

[DeMe94] M. J. Dent, R. E. Mercer. Minimal forward checking. In 6th International Conference on Tools with Artificial Intelligence, New Orleans, USA, 1994.

[FBRe94] E. C. Freuder, C. Bessière, J. C. regin. Using inference to reduce arc consistency computation. In Proceedings of ECAI'94, Amsterdarm, The Netherlands, 1994.

[GAS 97] Gérard Assayag, Carlos Agon, Joshua Fineberg, Peter Hanappe. An Object Oriented Visual Environment For Musical Composition. Proceedings of the ICMC 97, Thessaloniki, 1997.

[Mack77] A. Mackworth. Consistency in networks of relations. Artificial Intelligence, 8:99-118, 1997.

[Pratt92] V. Pratt, The Duality of Time and Information. Proceedings of CONCUR'92. Springer-Verlag, New York, 1992.

[RuVa97] Camilo Rueda, Frank Valencia. Improving forward checking with delayed evaluation. In proceedings of CLEI'97, Santiago, Chile, 1997.

[SCO 98] CMN by Bill Schottstaedt, http://ccrma-www.stanford.edu/CCRMA/Software/cmn/cmn.html

[Steel90] G.L. Steele. Common Lisp The Language. 2nd Edition. Digital Press. 1990.

[VaRu96] Frank Valencia, Camilo Rueda. Uso de deducciones de no viabilidad en el calculo de arco consistencia. In Proceedings of CLEI96, Bogota, Colombia, 1996.