OMChroma: Compositional Control of Sound Synthesis

Carlos Agon, Jean Bresson, Marco Stroppa

OMChroma: Compositional Control of Sound Synthesis

Carlos Agon[1], **Jean Bresson**[1], and **Marco Stroppa**[2]
[1]STMS Lab: IRCAM / CNRS / UPMC, Paris, France.
[2]University of Music and the Performing Arts, Stuttgart, Germany

This is the authors' pre-print version of the article published in *Computer Music Journal*, 35:2 (2011). The published article is available at https://www.mitpressjournals.org/doi/10.1162/COMJ_a_00057

* * *

OMChroma is a framework for sound synthesis integrated into the computer-aided composition environment OpenMusic [1, 3]. It is a generalization of *Chroma*, a system developed by Marco Stroppa in the early 1980s. *Chroma* was initiated at the *Centro di Sonologia Computazionale* of the University of Padua. At that time, it mainly consisted of PLay First- (PLF-) routines written in Fortran for the Music V synthesis language [13]. The project was then generalized and written in LeLisp and Csound at the Massachusetts Institute of Technology between 1984 and 1986. Finally, starting in 1989, it was ported to Common Lisp and later integrated into OM at the *Institut de Recherche et de Coordination Acoustique/Musique* (IRCAM) with the help of Ramón González-Arroyo, Jan Vandenheede, and especially Serge Lemouton. Since then it has been extensively used for all of Marco Stroppa's pieces with electronics.

The main purpose of OMChroma is to devise appropriately expressive concepts to deal with the large amounts of data required by sound synthesis. For this reason, the layers of control and digital signal processing (DSP) are separated and connected by means of abstract structures internally interpreted for a given software synthesizer. Sound synthesis processes are considered from the standpoint of banks of functionally identical DSP units (e.g., oscillators, *fonction d'onde formantique* [FOF] synthesis, distortion modules, samplers, filters, envelope generators). OMChroma handles the control of these banks through a general structure similar to a bi-dimensional matrix, where rows and columns set up vectors of values for identified parameters of the DSP units. As OM provides a powerful framework for composition, the implementation of Chroma in this environment allows efficient extension of computer-aided compositional tasks to the layer of microstructural composition needed in software synthesis.

After a general introduction to the concepts and motivations of OMChroma, we will describe how its control structures can be used and dynamically defined in OM, and introduce higher-level structures for handling temporal and spectral data.

Introduction to the OMChroma Concepts

Although OMChroma is not bound to any specific synthesizer, current work mainly focuses on the control of Csound (Boulanger 2000) for historical reasons, but also for the computational and expres- sive power, widespread usage around the world, and the amount of knowledge and examples available with this language.

Control Strategies for Sound Synthesis

Let us introduce the OMChroma concepts with a simple example. Suppose that a composer's idea is to generate a harmonic sound containing five partials with different amplitudes. To be carried out, this idea has to be implemented into a "synthesis process," defined as a set of DSP units and their control data. Three possible implementations are discussed herein — see also [29].

In Csound, as in earlier "Music *N*" languages [18], a synthesis process is divided into two parts: an *orchestra*, including one or several numbered *instruments*, and a *score*, containing a sequence of instructions (or *statements*) activating the instruments at different times and with different parameters. The communication between the instruments and the score is made via *p*-fields: the symbol *pn* in an instrument represents the parameter instantiated by the *n*th value of a corresponding score statement.

Implementation Using a Bank in the Audio Table

Our first implementation uses two table look-up oscillators (poscil), one for the amplitude envelope and one for the audio. The audio table is made of five harmonics with different relative amplitudes, declared via the Csound *GEN10* table generator. The instrument and score are as follows:

```
INSTRUMENT:
instr 1
  ; amplitude envelope
  env poscil p4, 1/p3, p6
  ; sine tone (freq = p5)
  out poscil env, p5, 1
endin

SCORE:
  ;;; TABLES DECLARATION
  ; 1: wave table
  f1 0 32769 10 1 0.3 0.6 0.2 0.45
  ; 2: amp envelope
  f2 0 4097 7 0 500 1 1000 0.5 1000 0.5 1597 0

  ;;; SCORE STATEMENT:
  ;;; ACTIVATING THE INSTRUMENT
  ; p1=inst p2=onset p3=dur
  ; p4=gain p5=freq
  ; p6=amp env. (table num)
  i1 0 10 0.5 110.0 2
```

This is an efficient implementation (two oscillators generate everything), but also quite a limited one. Sounds with a different number of harmonics may be generated by changing the audio oscillator's wave table ($f1$), but all of these harmonics will have the same duration and amplitude envelope, determined in the score statement.

Implementation Using a Bank in the Orchestra

Instead of hard-wiring the components inside of a table, we can implement this process using five sine-tone generators. The instrument then turns into a bank of oscillators whose individual input parameters have to be determined in the score:

```
INSTRUMENT:
instr 1
  env1 poscil p4, 1/p3, p14
  part1 poscil env1, p5, 1
  [...]
  env5 poscil p12, 1/p3, p14
  part5 poscil env/5, p13, 1
  out part1 + part2 + part3 + part4 + part5
endin

SCORE:
  ;;; TABLES DECLARATION
  ; 1: wave table
  f1 0 32769 10 1
  ; 2: amp envelope
  f2 0 4097 7 0 500 1 1000 0.5 1000 0.5 1597 0

  ;;; SCORE STATEMENTS:
  ; p1=inst p2=onset p3=duration
  ; p4=gain1 p5=freq1 p6=gain2 p7=freq2
  ... p12=gain5 p13=freq5
  ; p14=amp envelope (table num)
  i1 0 10 1 110 0.3 220 0.6 330 0.2 440 0.45 550 2
```

This implementation, though less efficient, allows us to generate non-harmonic sounds by freely modifying the input frequencies. Its main drawback is that the maximum number of partials is fixed in the instrument. Moreover, if more partials were needed, the instrument and score would also become very cumbersome to read and edit. Although separate amplitude envelopes or additional control parameters might be added, all of them would still have the same onset and duration because they are activated by a common score statement.

Implementation Using a Bank in the Score

Finally, if the instrument contains a single DSP unit, the bank can be entirely created in the score: each sine tone corresponds to one score statement.

```
INSTRUMENT:
instr 1
  env poscil p4, 1/p3, p6
  out poscil env, p5, 1
endin

SCORE:
  ;;; TABLES DECLARATION
  ; 1: wave table
  f1 0 32769 10 1
  ; 2: amp envelope
  f2 0 4097 7 0 500 1 1000 0.5 1000 0.5 1597 0

  ;;; SCORE STATEMENTS:
  ; p1=inst p2=onset p3=dur p4=gain p5=freq
  ; p6=amp envelope (table num)
  i1 0 10 1 110.0 2
  i1 0.2 10 0.3 220.0 2
  i1 0.1 10 0.6 330.0 2
  i1 1.05 10 0.2 440.0 2
  i1 1.0 10 0.45 550.0 2
```

Although this implementation is the least efficient, because multiple instances of the instrument are required, it is the most adaptable: the dynamic allocation of the different instances of the instrument makes it possible to set an independent frequency, amplitude, envelope, duration, and onset time for each individual partial and for any number of partials.

To summarize, even if the three implementations produce the exact same signal, they not only present various levels of efficiency and flexibility, but also differ in opportunities for experimentation and expansion. The same signal could also be produced by carefully setting the parameters of other synthesis techniques. Each implementation, however, is unique in regard to how the original idea can be developed, that is, its *sound potential*.

The Sound Potential

Despite their possible sonic equivalence, the implementations described here are very different if examined from a compositional standpoint. If they were to be developed, the three processes could not be modified in the same way. As is clear when comparing the scores, they are not driven by the same conception nor controlled by the same data structures. We say that they do not belong to the same "epistemological paradigm." The idea of "sound potential" [8, 29] is based on such conceptual distinctions regarding sounds created in compositional contexts.

A sound potential represents the potentially infinite collection of all the sounds generated by a given set of processes, and recognized as belonging to the same compositional idea. This compositional idea may be "static," producing different sounds that are considered as a sort of "variation." A bell sound with different spectral centers of gravity, durations, or "inharmonic" properties is an example. On the other hand, the compositional idea may be "evolutive," that is, embedded into a development that changes the nature of the sound over time. A "sound potential" is therefore quite different from a "sound object" in the tradition of Pierre Schaeffer [23]. The latter is, to put it simply, a phenomenological description of an existing recorded or processed sound, whereas the former is a specific morphological concept defining a set of cognitively consistent sounds.

As McAdams shows [15], recognizing a sound implies a preliminary perceptual process, but is, fundamentally, a higher conceptual task. From this perspective, a sound potential shares some of the structural features of Stroppa's "musical information organisms": time-evolving morphologies partially inspired by the work of E. E. Smith [16] and E. Rosch [22, 21] — see also [28]. The task of sound recognition, of course, depends on the musical context. For example, recognition depends on how and in which order a set of sounds is used, as well as on the familiarity of the listener with the material. This is, therefore, at the same time a compositional, aesthetic, and technical issue: Each sound potential embeds a certain knowledge about sonic processes; that is, a certain way a composer thought about and implemented the sonic processes. Technically, this knowledge is expressed by some rules and constraints about how single sounds are to be produced and developed in order to satisfy the constraints of recognition, even if the final criteria of discrimination are more compositional than merely perceptual. It is essential that an environment dedicated to the control of sound synthesis provides powerful and expressive tools to implement this kind of behavior.

Generalized Control Representation in OMChroma: The Matrix

The control of sound synthesis processes in OMChroma is an extension of the third case discussed earlier, where the control structure, the score, generates banks of parameters for a DSP unit. The Csound score, in this case, contains as many statements as there are elements in the bank, and looks like a matrix, where columns correspond to the different parameters (*p*-fields) and rows to different instances of the instrument (see Table 1).

The control structures in OMChroma are also matrices. They correspond to particular synthesis units and contain (1) a fixed number of rows (as many as there are parameters, or *p*-fields, in the Csound instruments), and (2) a variable number of columns corresponding to the score statements (also called *components* in the matrix). In other words, OMChroma "turns" the Csound rows into columns and vice-versa (see Table 2). Each row, therefore, indicates the variation of a parameter across the whole bank: When the matrix is read (or interpreted) vertically, column by column, the score lines can be reconstructed.

Note that in Table 1, some data vary in every row, while other do not. In the OMChroma matrices (see Table 2) the values can be specified without necessarily repeating those that do not change (e.g., rows *Duration* and *Amp. Env.*) As visible in Table 2, the row's contents can also be set using functional descriptions (see rows *Onset* and *Frequency*).

Instr. Num (p1)	Onset (p2)	Duration (p3)	Amplitude (p4)	Frequency (p5)	Amp. Env. Num. (p6)
i1	0	10.0	1.0	110.0	2
i1	0.2	10.0	0.3	220.0	2
i1	0.1	10.0	0.6	330.0	2
i1	1.05	10.0	0.2	440.0	2
i1	1.0	10.0	0.45	550.0	2

TABLE 1: *Matrix Representation of a Csound Score.*

Component Num.	1	2	3	4	5 (*n*)
Onset (p2)	*[random onset between 0.0 and 1.2]*				
Duration (p3)	*10.0*				
Amplitude (p4)	1.0	0.3	0.6	0.2	0.45
Frequency (p5)	*[nth harmonic of 110.0]*				
Amp. Env. Num. (p6)	*2*				

TABLE 2: *Reversed Matrix in OMChroma with Higher-Level Specification Rules.*

It is useful to emphasize here that OMChroma's typical use of Csound scores is different from the meaning of a score in the usual musical sense, and, perhaps, also different from Max Mathews's original vision of Music *N* scores. In OMChroma, the scope of a Csound score (i.e., the different "lines" corresponding to the matrix lines) is very low level. The high-level musical concept of a score, whose scope is a composition, is handled at the level of OpenMusic. There is then a high-level score (a composition specified as an OpenMusic program) that includes many low-level, usually very long, Csound scores. It is also often the case that OMChroma generates several relatively short sounds, which are then either mixed in an appropriate environment or directly performed in a concert. As a consequence, a Csound score generated by OMChroma typically specifies the components of a single sound event, rather than the sound events of a whole composition.

Virtual Synthesizer

Although the current examples and actual implementation mainly focus on Csound, OMChroma aims at representing and controlling the parameters of sonic processes independent of a given syn- thesizer, synthesis engine, or computer platform. This is achieved by creating abstraction barriers between the control layer and the data needed by each synthesizer. The matrix corresponds to this abstraction level: The control layer lies in the domain of compositional techniques (it is the "composer" of the system), and the translation of the internal data for a specific synthesizer can be seen as the "interpreter."

A special effort was made to provide a consistent taxonomy of control parameters. By adopting a common naming convention across different synthesis techniques, users could gain experience in handling these parameters and develop an "inner ear" for the control of sound synthesis. This semantic consistency makes it possible to easily interpret and trans- late the data of a given matrix, so as to drive another instrument (provided it has semantically equivalent inputs) without changing the control structures and strategies. This led the authors to introduce the idea of a "virtual synthesizer" in the conceptual design of OMChroma.

Visual Programming Framework

OM is a computer-aided composition environment where processes can be expressed in the form of visual programs (*patches*) made of functional boxes and connections. It is a complete functional and object-oriented visual programming language based on CLOS [9]. Musical data are represented by objects generated via *factory* boxes, boxes that create and initialize instances of classes (e.g., notes, sequences, rhythms), and

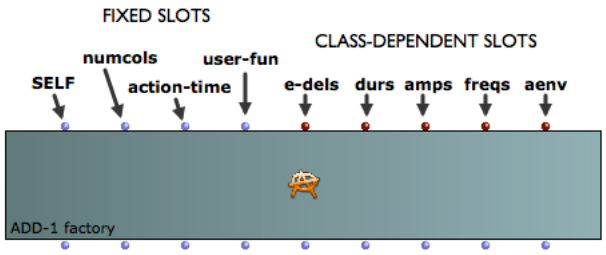

FIGURE 1: *Example of an OMChroma class factory: ADD-1.*

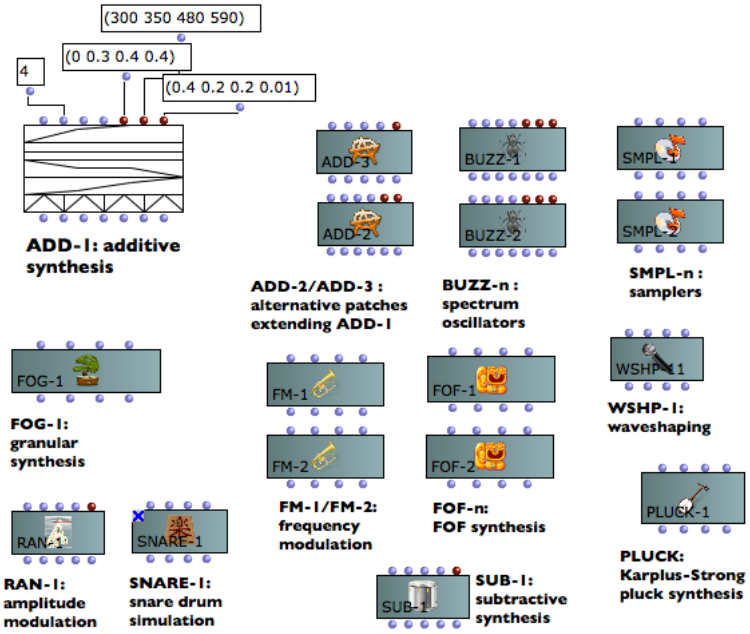

FIGURE 2: *The main basic OMChroma classes.*

whose inputs and outputs (represented by small round inlets and outlets) are connected to the internal slots (or attributes) of these classes. Factories are generally associated with an editor, which permits the inspection and modification of the last-created instance. The contents of this editor can be displayed in the box itself and made visible in the patch.

OMChroma Classes

An abstract class named *class-array* was added to the OM kernel in order to represent matrix structures [2]. This class is mainly intended to be sub-classed: In OMChroma, the *class-array* subclasses are associated with specific synthesis instruments, and their additional slots correspond to the different parameters of these instruments. The rows of the matrices are, therefore, determined by the different slots of the class (the synthesis parameters), and the number of columns is given by a global slot of the *class-array*. A reference to the original DSP unit (the Csound instrument code) is internalized in the class (as a static or "class allocation" slot), and will be used to synthesize sounds from instances of this class.

Figure 1 shows a factory box for the class *ADD-1*, a subclass of *class-array*. *ADD-1* is the simplest OMChroma class; it corresponds to the process (or sound potential) discussed in the previous section.

All of the OMChroma classes (including *ADD-1*) have four fixed global input and output slots (see Figure 1). From the left, the first one (*SELF*) represents the instance produced by the factory box, as is the case with all the OM factory boxes. The second one (*numcols*) represents the number of columns (or components) in the matrix. The third and fourth ones (respectively labeled *action-time* and *user-fun*) will be discussed later. The other inputs and outputs visible in the box correspond to the different rows of the matrix (the parameters of the synthesis instrument). They are class-dependent slots, although some

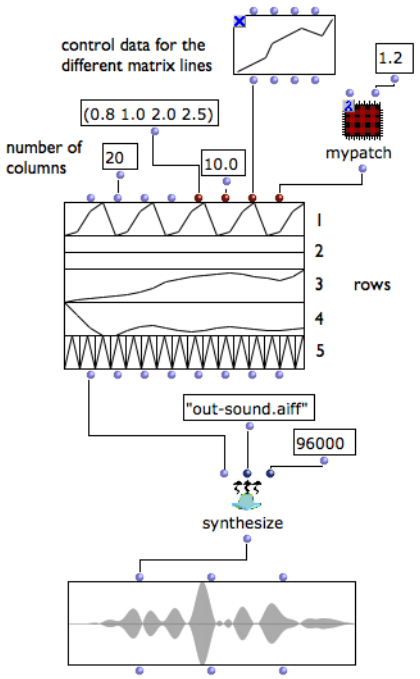

FIGURE 3: *Sound synthesis in OMChroma.*

will be found in every class, such as onsets (entry delays or *e-dels*) or durations (*durs*).

Following this model, a large library of classes is available in OMChroma. As shown in Figure 2, these classes correspond to various synthesis techniques (more complex sine generators, Karplus-Strong synthesis, frequency modulation, snare simulators, FOFs, samplers, granular synthesis, etc.).

Dynamic Class Definition

In addition to the default synthesis classes, new classes can be dynamically created in OM from any working Csound orchestra. The instruments in this orchestra are detected and parsed, and their different parameters (*p*-fields) are matched to the slots of a new class.

Visual, object-oriented programming features in OM allow users to complete or refine the class definition: Class slots can be renamed, set to particular types (such as a number or a function), and assigned default values. A predefined OM class called *cs-table* is used to define and declare Csound tables using break-point functions. Inheritance also enables specifying particular (and possibly multiple) super-classes for the new class [2]. The original Csound code attached to the class can also be edited afterward, provided that the number of *p*-fields remains equal to the number of slots.

Sound Synthesis

The *synthesize* function implements the concept of virtual synthesizer by converting matrix instances into sounds. In Figure 3 the contents of the *ADD-1* factory box is visible and filled with control data. The matrix is connected to the *synthesize* box, which internally generates the corresponding Csound instrument and score files. A new sound file is then synthesized via Csound and loaded in the sound box at the bottom of the figure.

Synthesize accepts additional parameters, such as a sampling rate, output file name, etc. (see Figure 3), and also global variables, tables, or macros to be declared and used in the synthesis process. It is capable of synthesizing various matrices simultaneously (supplied as a list), and possibly of different types (several instruments will be gathered in a Csound orchestra and controlled individually in the score).

Symbolic Control and Matrix Instantiation

The instantiation of an OMChroma class is performed by providing values for the different rows of the matrix. The number of required values (the number of components or columns in the matrix) is determined by the slot *numcols*. Several types of data can be used to set these values

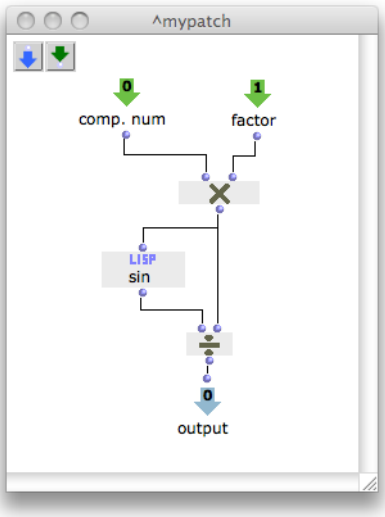

FIGURE 4: *Contents of the patch box "mypatch" from Figure 3. This visual program corresponds to the Lisp expression* (lambda (i x) (/ (sin (* i x)) (* i x))), *where* i *is the component number and* x *is a factor (whose value in Figure 3 is 1.2).*

(see Figure 3): lists of numbers (if shorter than the number of components, the values are repeated, if longer, the unused values are ignored); single values applied to all of the components; break-point function (*BPF*) objects, sampled according to the number of components as an envelope applied to the whole matrix line (first and last values are preserved and the intermediate values are interpolated and equally distributed across the components); and functions (Lisp functions or OM "lambda" boxes with one argument) evaluated at each successive component. Figure 4 shows the contents of the lambda box "mypatch" from Figure 4 (a small lambda icon is visible at the top-left corner of the box), which is connected with and specifies the contents of row #4 in the matrix. This box defines a lambda function with one argument that corresponds to the single "free" input of the box and is bound to the indices of the successive components.

All input data and functions can be the product of other functions or programs. Specific types of data (for instance, *cs-tables* for the amplitude envelopes) can be declared for the class's slots and are converted accordingly. Finally, all the slots of a given class are not necessarily visible in the factory box, reducing the number of (possibly) unnecessary parameters to handle. The box in Figure 3, for example, shows only four of the five available parameters of *ADD-1*: Row #5 is therefore automatically filled with a default value for the slot *aenv* (amplitude envelope).

External Controls and Cross-Parametric Relations

Another particularity of the *class-array* is the possibility of interconnecting input parameters among each other: A list can be connected to a given matrix input slot, whose first element is a function followed by its arguments, which may be either values or strings designating other slots.

Additional rows, called *controls*, can also be created to represent external control values that do not belong to the actual parameters needed by the instrument.

Figure 5 shows an example of both a *control* row and a cross-parametric relation. One of the slots (*freq*) is set with the list (*mypatch*, *"control-1,"* *"durs"*). Here, *control-1* is a control slot, and *durs* is a regular slot of the class. The values for *freq* will be computed by applying the function defined in *my-patch* to the values of *control-1* and *durs* for each component. The contents of *mypatch* is open and visible on the right of the figure.

Data Processing and Delayed Parsing: The "user-fun"

Once a matrix is instantiated, its internal data can still be accessed and modified thanks to a graphical editor. This manual access, however, is very unwieldy for large structures. More importantly, access can be performed in a program via a set of dedicated functions (*get-comp*: get component *i*; *comp-field*: read/write a component's given field; *new-comp*, *remove-comp*: create or remove a component; etc.).

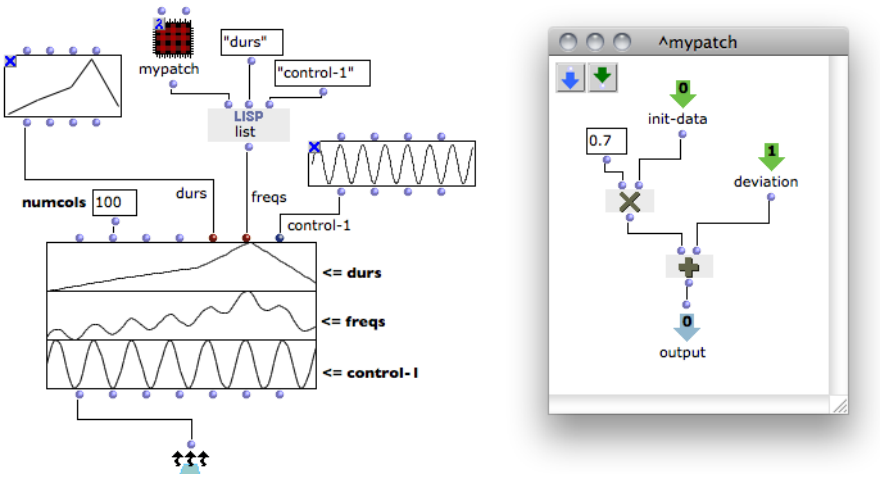

FIGURE 5: *Cross-parametric relations: the frequency* (freq) *of each component is computed as a function of its duration* (durs) *and of an additional deviation specified in* control-1.

These functions are applied only after the matrix has been instantiated. In particular, they can step in during the *synthesize* process and behave as delayed processing tools to modify the initial matrix. To do so, they must be attached to the matrix via the global slot *user-fun* (see Figure 1). The "user-fun" is a global function called on every component just before the matrix is converted into actual synthesis parameters. It can access the current matrix data and adjust, filter, compute, modify, remove, or add components. The added or modified components do not appear in the visible matrix, but only in the final score. This system permits rule-based control of sound synthesis (see, for instance, the PLF-routines in Music V, or the "user-routines" for the control of CHANT [19]).

As always in OM, this function can be written either textually (in Lisp) or graphically (in a visual program). Here is an example of a simple function that tests the frequency of each component and removes it if it is above the Nyquist frequency:

```
; discards component c if freq[c] > SR/2
(defmethod fq-sr? ((c component))
  (let ((sr2 (/ *curr-sample-rate* 2)))
    (if (> (comp-field c ''freq'') sr2)
      (remove-comp c)
     c)))
```

Higher-Level Control Processes

In this section we present various ways to embed the aforementioned structures in more advanced control procedures and higher-level time structures.

Clusters and "Sub-Components"

Psychoacoustic research and practical experience have shown the importance of slight deviations in the synthesis parameters, such as jitter, beatings, and vibrato, to obtain sounds that are perceptually more natural, fused, and musically better — see Risset's ex. 430 [17] (three successive approximations of a bell-sound), [14], or [31]. Of course, these deviations could be included in the Csound orchestra file, but their scope and flexibility would remain limited. The strategy emphasized in OMChroma is, again, to enrich and extend the scores.

For instance, it is possible to consider each individual component of a matrix as a "micro-cluster," that is, as sets of "sub-components" centered around it and extending its musical semantics.

Figure 6 shows a graphically defined function connected as a "user-fun" that generates micro-clusters: Each component is completed with a more or less dense set of sub-components randomly spread around its frequency. Two *control* slots have been added to the matrix: *npart* determines the number of additional sub-components and *ston* the mistuning (or *stonatura* in Italian). Their values describe the cluster-related parameters (i.e., density and spread), which can vary across the different components of the matrix. Depending on these values, resulting sounds will be enriched with additional spectral components generating more or less regular beatings, or larger-scale "chords" perceived as perturbations of the initial frequency contents.

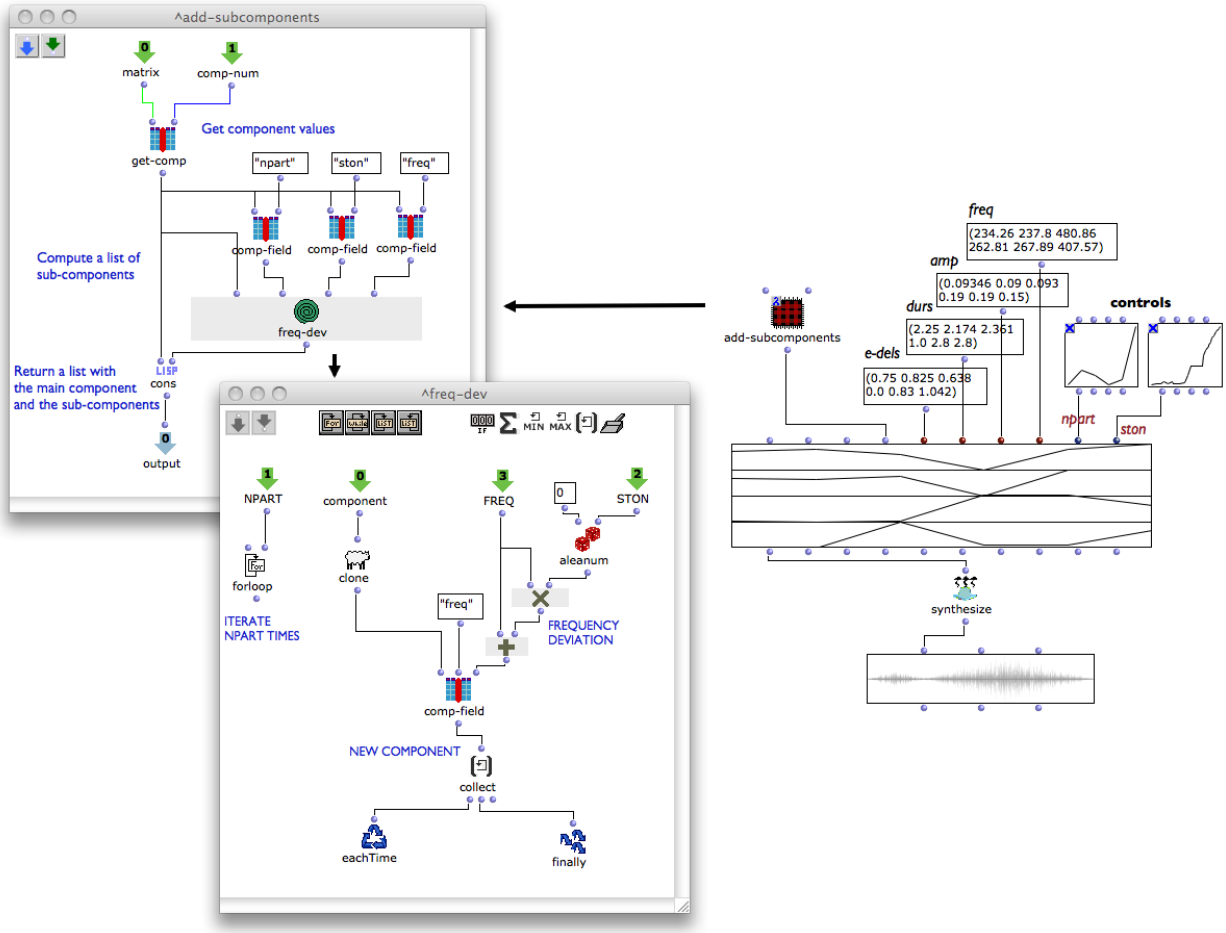

FIGURE 6: *Generating clusters for each component using the* user-fun *and* control *slots.*

The Csound score listed here demonstrates the evolution of the density and spread of the sub-components for the successive clusters (a similar process is applied to the entry delays, durations, and amplitudes).

```
; component 1: 0 sub-components
i1 0.75 2.25 0.09346 234.26 500
; component 2: 3 sub-components
i1 0.825 2.174 0.09 237.8 500
i1 0.901 2.098 0.09 244.52 500
i1 0.977 2.022 0.09 241.09 500
i1 0.872 2.427 0.88 252.83 500
; component 3: 2 sub-components
i1 0.638 2.361 0.093 480.86 500
i1 0.704 2.295 0.093 591.3 500
i1 0.694 2.605 0.83 767.78 500
; component 4: 0 sub-components
i1 0.0 1.0 0.19 262.81 500
; component 5: 1 sub-component
i1 0.83 2.8 0.19 267.89 500
i1 0.8 2.8 0.15 417.19 500
; component 6: 12 sub-components
i1 1.042 2.8 0.15 407.57 500
i1 1.045 2.8 0.16 450.27 500
i1 1.048 2.8 0.10 455.13 500
i1 1.042 2.6 0.16 410.30 500
...
```

The non-deterministic aspect in this delayed processing also implements a sort of interpretation scheme: Each time the matrix is synthesized, the score is slightly different, and therefore never produces an identical sound, though it corresponds to the same "sound idea."

Matrices as Events

The notion of event is crucial in the compositional approach to sound synthesis. What "event" means for a composer, or how large, efficient, and expressive it should be, is a constantly discussed issue, and therefore requires high levels of flexibility from musical environments (see for instance [10] or [6] for related discussions). An "event" can generally be defined as a composer's "conceptual primitive," that is, a structure implementing a compositional thought that does not require further decomposition. This is still, however, a highly variable notion dependent on the composer's aesthetics and formal strategies.

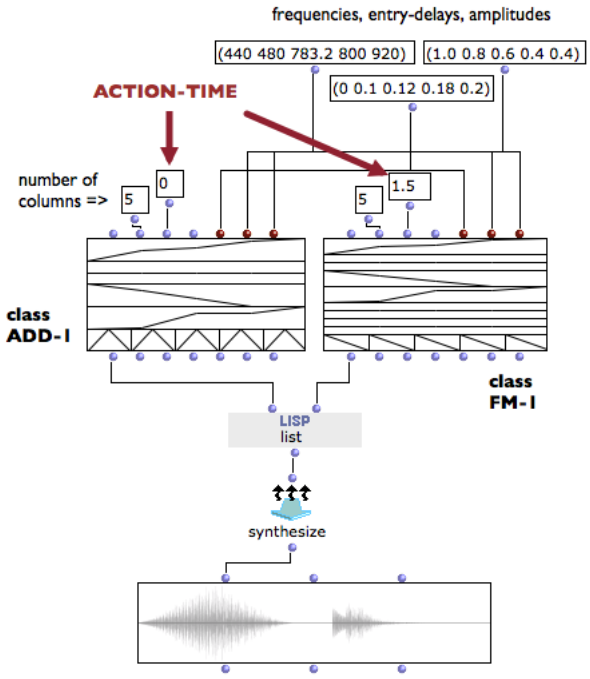

FIGURE 7: *Synthesizing a list of timed matrices.*

Being the interface between the symbolic world and the "real" synthesizer, the OMChroma matrix embeds this notion of primitive event. Even if any attempts to categorize a "compositional primitive" are bound to be very schematic, being aware of this notion may help find the best strategy when Figure 7. Synthesizing a list of timed matrices. formalizing a sound process. OMChroma was designed so as not to impose any particular approach to the composer's choice: The matrices can be considered either as container objects, including an internal temporal organization (where each internal component, or microcluster, is regarded as an event), or as primitive events of their own, integrated into larger temporal organizations.

So far, they encompass a temporal dimension via two common slots: The entry delays (*e-dels*) and the durations (*durs*). Another general slot, *action-time* (see Figure 1), represents a global time delay applied to all components. When synthesizing a list of matrices it is, therefore, possible to consider them as timed events whose onset is determined by their *action-time*. Figure 7 illustrates this situation, with two matrices and two different *action-times*: Each matrix generates an "event" in the resulting sound. In this example, the events are clearly separated (t1 = 0 sec, t2 = 1.5 sec), but they could also overlap, be linked in more intricate ways, and be computed from any program or external temporal structure. (Figure 7 also illustrates the possibility to perform synthesis with classes, other than *ADD-1* and *FM-1*, that are instantiated with similar control data.)

From our experience, several compositional approaches can be identified with respect to events: (1) a part of a sound (where the perceived sounds are actually mixes of several eventsthis technique was extensively used in the composition of Stroppa's *Traiettoria* (1982–1986) [26]; (2) a sound of its own with no obvious internal temporal or rhythmical structure; (3) a "phrase" (or sound process) that is perceived as a stream of sound sources; (4) a "section," where the sound process has a complete formal structure, often resulting from some kind of algorithmic approach; and (5) a "template": any of the previous cases condensed into a synthetic form and expanded by a *user-fun*.

Two advanced approaches to handle temporal structures from events are discussed in the next sections.

Maquettes as Temporal Structures

The OM *maquette* extends the notion of visual program to the temporal dimension, and provides a way of integrating synthesis events within a time structure. A maquette contains musical objects and programs represented by boxes, connected by temporal and functional relations. An extension, introduced in [5], allows the system to per-

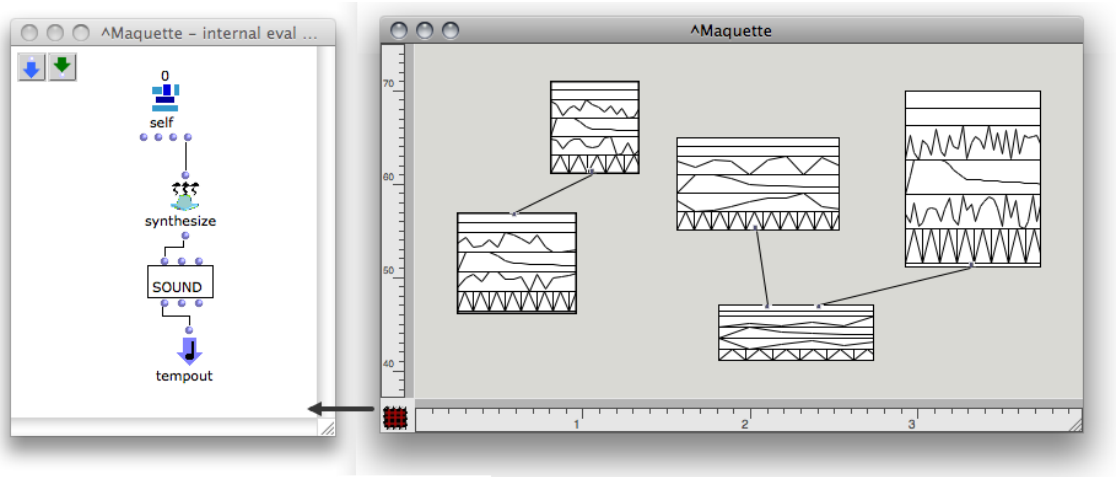

FIGURE 8: *Using the* maquette *as a temporal structure and context for the synthesis process.*

form sound synthesis processes in this temporal framework, where each box in the maquette is a program generating synthesis parameters (see Figure 8). Any of these parameters may depend on internal processes, on properties of their containing box (size, position, etc.), or on external data coming from other boxes.

While each matrix is assigned an onset depending on its position, a global process (here, the *synthesize* function in the left-hand part of the figure) computes a sound from the values and overall temporal configuration of the boxes. The maquette editor allows users to visualize and control global structure; it is the external representation of this program unfolded in time.

Chroma Models

As an alternative to maquettes, and containing temporal information at a higher level than the matrix, OMChroma provides Chroma models. *Chroma models* are abstract data structures used as reservoirs of forms and parameters for sound synthesis. They are made of a time structure (a list of time segments) and a sequence of Vertical Pitch Structures (VPS).

VPS are polymorphic structures devised by Stroppa for his earlier works [27]. They represent spectral material as absolute, relative, or pivot-based symbolic pitches or frequencies, and are meant to bridge the gap between a symbolic melody- or harmony-oriented approach to composition and numeric, spectral materials.

These concepts have been implemented in OMChroma [7] as, respectively, the *cr-model* object and a set of other classes corresponding to the main categories of VPS.

When building a *cr-model*, the first step is to specify a collection of time markers and some frequency information, which can originate from either programs or sound analysis procedures (sonograms, formant or transient detection, partial tracking, etc.). Analysis data are stored as SDIF files [25], and loaded as *SDIFFile* objects in OM (see the box labeled "sndanalysis.trc.sdif" at the upper-left of Figure 9). The *cr-model* internally segments these data using the time markers and converts the successive segments into a list of VPS.

The editor windows (see the right side of Figure 9) display the contents of the *cr-models* (a list of pitches with frequency on the vertical axis) divided in time segments (on the horizontal axis). The values in the upper-right part indicate the time and frequencies corresponding to the current mouse position in the window, and the leftmost vertical frame displays the energy distribution of the different pitches of one selected segment of the sequence.

The data collected in the first *cr-model* in Figure 9 could be represented as follows:

intervals (sec.)	**pitch list / amplitudes in the VPS**
[0.0-1.0]	146.49 221.11 330.77 371.27 441.73 ...
	0.0900 0.0572 0.0346 0.0756 0.0263 ...
[1.0-3.0]	146.86 221.43 330.64 372.35 443.71 ...
	0.0904 0.0742 0.0141 0.0412 0.1533 ...
[3.0-6.0]	115.41 184.58 231.60 276.23 373.85 ...
	0.0666 0.0332 0.0230 0.0265 0.0875 ...
[6.0-9.0]	123.10 219.81 243.37 292.85 370.93 ...
	0.0947 0.0461 0.0231 0.0396 0.0920 ...
...	...

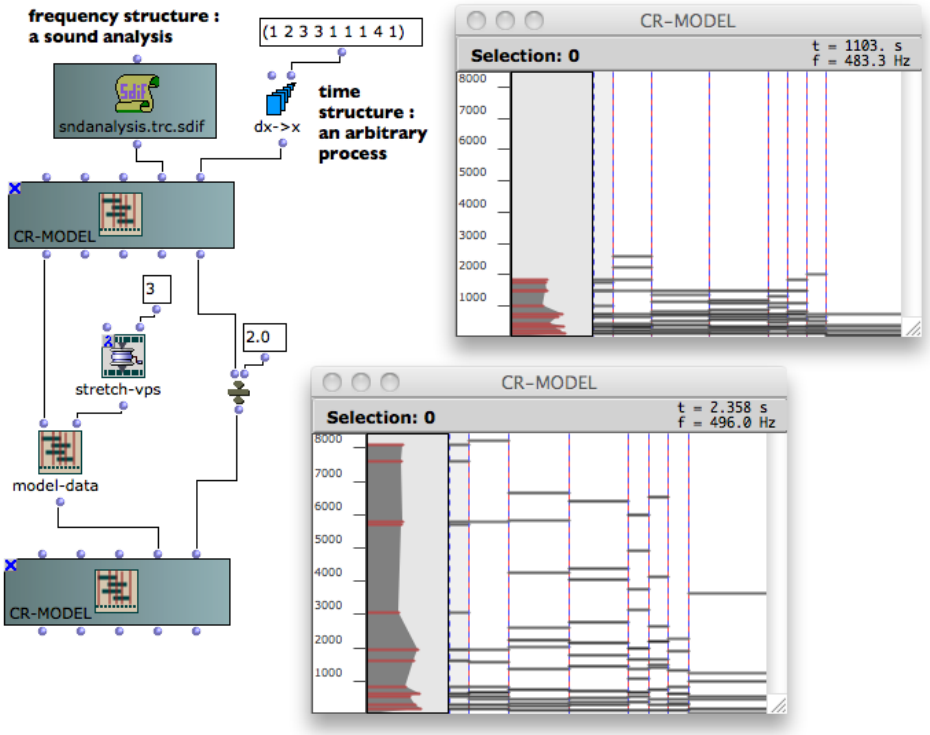

FIGURE 9: CR-MODEL*: Representation of abstract sound models in OMChroma.*

Because the frequency and time structures are independent, a *cr-model* can be constructed from any arbitrary combination of them. The purpose of a model, even when derived from sound analysis data, is not to reconstitute realistic reproductions of an original sound, but rather to explore the data's sound potential. The compositional aim is to provide structured characteristics that both grasp the organic liveliness of "real" sounds and are abstract enough to be applied to any kind of synthesis process.

The *cr-models* can be transformed in order to yield personalized structures more or less related to the original ones. In Figure 9, the *model-data* function box gets the data from the first *cr-model* (the VPS list). Its second input applies one or more auxiliary functions to each VPS collected. Some predefined functions are available, such as filters, transpositions, frequency shifting, and stretching. User-defined functions can also be programmed in order to specify particular behaviors. Time structures may be modified as well, and independently of the frequency domain (permutation, compression, stretching, or transformations based on symbolic rhythms, etc.).

The transformed data used to instantiate the second *cr-model* in Figure 9 are partially reported in the following listing. As can be read in this listing, the time structure has been compressed (times have been divided in half) and the frequency content has been stretched. The stretching function used in this example (*stretch-vps*) is a predefined function inspired by [14] that generates different kinds of distorted spectra by stretching or compressing the frequencies according to their relative intervals, without modifying the value of the lowest one in the list (the fundamental frequency).

intervals (sec.)	**pitch list / amplitudes of the VPS**
[0.0-0.5]	146.49 281.32 532.64 639.64 842.46 ... 0.0900 0.0572 0.0346 0.0756 0.0263 ...
[0.5-1.5]	146.86 281.55 443.11 531.51 641.65 ... 0.0904 0.0742 0.0141 0.0412 0.1533 ...
[1.5-3.0]	115.41 242.94 348.08 460.25 743.54 ... 0.0666 0.0332 0.0230 0.0265 0.0875 ...
[3.0-4.5]	123.10 308.56 362.60 486.19 707.14 ... 0.0947 0.0461 0.0231 0.0396 0.0920 ...
...	...

At some point, the model data must be converted into matrices: For each segment of the time structure, *expand-model* (see Figure 10) translates the VPS into an instance of a given synthesis class. The resulting matrix list can then be synthesized via the *synthesize* box.

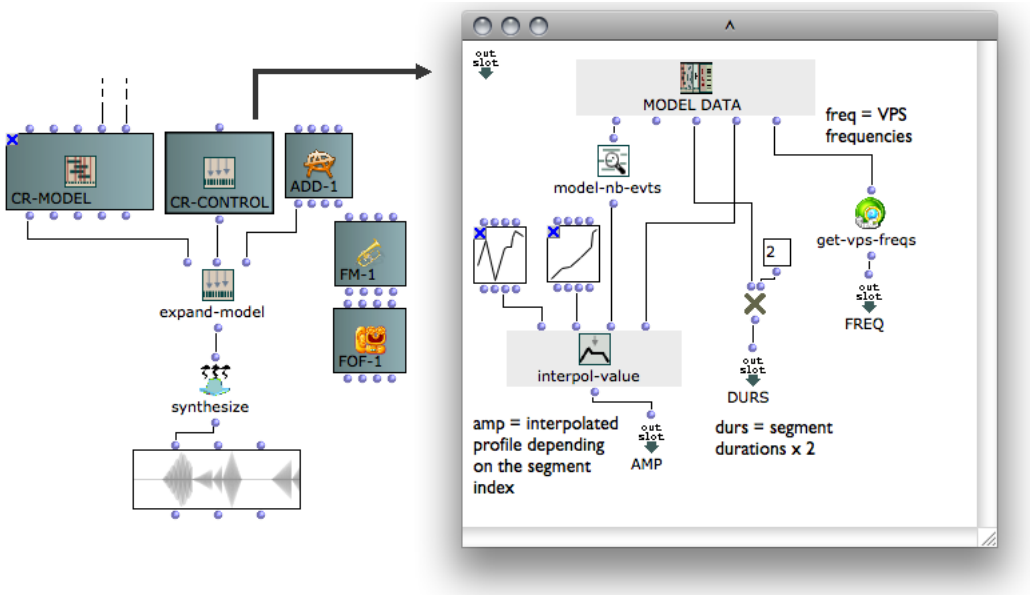

FIGURE 10: *Converting a* cr-model *to a list of matrices (via* cr-control*) and synthesis of a sound. The* cr-control *visible on the right implements a set of rules converting the VPS data from the* cr-model *to the different slots of a matrix class.*

The crucial task here is to specify the matching rules that generate the right parameters from the model data. Some rules are straightforward (e.g., sending frequencies and amplitudes to the corresponding slots), some rules may be more or less directly derived from the existing data, and other rules might need more advanced processing (psychoacoustic, mathematical, or any other personal rules). The *cr-control* object represents a set of matching rules that can be connected to *expand-model* to manage the transfer of data from a *cr-model* to OMChroma matrices.

Figure 10 shows rules defined in the *cr-control* editor (right-hand part of the figure). The *MODEL DATA* box in this editor represents the data for every segment in the model, and the three out- put arrow boxes (labeled *FREQ*, *AMP*, and *DURS*) correspond to the different slots of a matrix. In this example, the frequencies (*FREQ*) are directly extracted from the *cr-model*'s VPS list, while the amplitudes (*AMP*) are computed by an interpolation process independent from the VPS contents. The duration list of the matrices (*DURS*) are those of the *cr-model*'s time structure multiplied by 2, which will make the resulting successive events overlap while keeping their original onsets.

Rules of arbitrary complexity can be defined between the original model, the different slots of a class, and any sort of compositional processes. Because of the common naming convention adopted, and because the *cr-model* data are very abstract, they can be converted to instances of practically any class. It is generally possible to connect the contents of a *cr-model* to any matrix owning slots with common names while using the same *cr-control* matching rules (the type of the matrix is given by the third input of the *expand-model* box; for example, in Figure 10, the type is *ADD-1*).

Conclusion

OMChroma provides a complete framework to integrate sound synthesis and advanced compositional processes at various stages of sound conception. Due to the complexity of both the processes and the data involved, several strategies have been adopted: modularity in the design and usage of the synthesis processes (the synthesis class library and definition tools), high-level symbolic instantiation of these classes through programming and/or definition of behavioral rules, connections to sound analysis and synthesis in the compositional environment, and large-scale temporal modeling (the *maquette* or the *cr-model*).

OMChroma shares some features with systems dedicated to sound synthesis that were designed in the 1980s and 1990s, such as Formes [20], Common Music/Common Lisp Music [30], or Patchwork/PWGL [12, 11]. Like those systems, OMChroma's primary goal is not computing sound in real time during a concert, but providing an integrated environment where the composer can elaborate powerful and expressive control structures.

OMChroma's use of an external synthesis engine, and the modular design of the interface between the control layer and this synthesis engine, make writing extensions to other engines quite straight- forward to envisage. The underlying synthesis engine(s) can also be maintained, rewritten, and further developed independently of OMChroma. In addition, the integration of OMChroma into a computer-aided composition environment mainly dedicated to instrumental music provides symbolic and algorithmic richness. The gap between this instrumental approach and a synthesis-oriented approach is bridged, although each environment can also be used separately.

There are, of course, many other ways of thinking about sound from a compositional standpoint, which may be more or less well suited to the OMChroma conceptual framework. This system proved to be useful to composers with different aesthetic approaches, however, as shown by the recent interest from young composers such as Tolga Tüzün (*Metathesis*, for two double basses and electronics, 2006), Marta Gentilucci (*Radix Ipsius*, for ensemble and electronics, 2008; *exp.doc...et juv.*, for saxophone and electronics, 2010), and Sebastien Gaxie (*Le Bonheur*, electronic music for a film by A. Medvedkine [USSR, 1934], 2010).

Future work on this project will mainly concern the implementation of a more "continuous," "phrase-based" conception of time, as opposed to the current "event-driven" approach and data representation. In particular, the integration of the CHANT synthesizer [19] in this environment shall emphasize interesting issues about how this continuous conception can be tackled in OMChroma. Current research has also focused on sound spatialization and the introduction of spatial rendering in the OMChroma framework [24].

References

[1] Agon, C. 1998. *OpenMusic : un langage de programmation visuelle pour la composition musicale.* PhD Thesis, Université Pierre et Marie Curie, Paris.

[2] Agon, C., M. Stroppa, and G. Assayag. 2000. "High Level Musical Control of Sound Synthesis in OpenMusic." In *Proceedings of the International Computer Music Conference*, pp. 332–335.

[3] Assayag, G., *et al.* 1999. "Computer Assisted Composition at IRCAM: From PatchWork to OpenMusic." *Computer Music Journal* 23(3):59–72.

[4] Boulanger, R., ed. 2000. *The Csound Book: Perspectives in Software Synthesis, Sound Design, Signal Processing, and Programming*. Cambridge, Massachusetts: MIT Press.

[5] Bresson, J., and C. Agon. 2006. "Temporal Control over Sound Synthesis Processes." In *Proceedings of the Sound and Music Computing conference*, pp. 67–76.

[6] Bresson, J., and C. Agon. 2007. "Musical Representation of Sound in Computer-Aided Composition: A Visual Programming Framework." *Journal of New Music Research* 36(4):251–266.

[7] Bresson, J., M. Stroppa, and C. Agon. 2007. "Generation and Representation of Data and Events for the Control of Sound Synthesis." In *Proceedings of the Sound and Music Computing conference*, pp. 178–184.

[8] Cohen-Lévinas, D. 1993. "Entretien avec Marco Stroppa." In *Les Cahiers de l'IRCAM, 3: La composition assistée par ordinateur*. Paris: IRCAMCentre Georges-Pompidou, pp. 99–117.

[9] Gabriel, R. P., J. L. White, and D. G. Bobrow. 1991. "CLOS: Integrating Object-Oriented and Functional Programming." *Communications of the ACM* 34(9):29–38.

[10] Honing, H. 1993. "Issues in the Representation of Time and Structure in Music." *Contemporary Music Review* 9:221–238.

[11] Laurson, M., M. Kuuskankare, and V. Norilo. 2009. "An Overview of PWGL, a Visual Programming Environment for Music." *Computer Music Journal* 33(1):19–31.

[12] Laurson, M., V. Norilo, and M. Kuuskankare. 2005. "PWGLSynth: A Visual Synthesis Language for Virtual Instrument Design and Control." *Computer Music Journal* 29(3):29–41.

[13] Mathews, M., et al. 1969. *The Technology of Computer Music*. Cambridge, Massachusetts: MIT Press.

[14] McAdams, S. 1982. "Spectral Fusion and the Creation of Auditory Images." In M. Clynes, ed. *Music, Mind and Brain: the Neuropsychology of Music*. New York: Plenun Press, pp. 279–298.

[15] McAdams, S. 1993. "Recognition of Auditory Sound Sources and Events." In S. McAdams and E. Bigand, eds. *Thinking in Sound: The Cognitive Psychology of Human Audition*. Oxford: Oxford University Press, pp. 146–198.

[16] Nickerson, R. S., D. N. Perkins, and E. E. Smith. 1985. *The Teaching of Thinking*. Hillsdale, New Jersey: Lawrence Erlbaum Associates.

[17] Risset, J.-C. 1969. *An Introductory Catalog of Computer Synthesized Sounds.* Murray Hill, New Jersey: Bell Laboratories. Reprinted in the booklet of: Various Artists, 1995. *The Historical CD of Digital Sound Synthesis*, Computer Music Currents 13. Mainz: Wergo WER 20332.

[18] Roads, C. 1996. *The Computer Music Tutorial*. Cambridge, Massachusetts: MIT Press.

[19] Rodet, X., and P. Cointe. 1984. "Formes: Composition and Scheduling of Processes." *Computer Music Journal* 8(3):32–48.

[20] Rodet, X., Y. Potard, and J.-B. Barrière. 1984. "The CHANT Project: From the Synthesis of the Singing Voice to Synthesis in General." *Computer Music Journal* 8(3):15–31.

[21] Rosch, E., et al. 1976. "Basic Objects in Natural Categories." *Cognitive Psychology* 8:382–439.

[22] Rosch, E., and C. B. Mervis. 1975. "Family Resemblances: Studies in the Internal Structure of Categories." *Cognitive Psychology* 7(4):573–605.

[23] Schaeffer, P. 1966. *Traité des objets musicaux*. Paris: Seuil.

[24] Schumacher, M., and J. Bresson. 2010. "Spatial Sound Synthesis in Computer-Aided Composition." *Organised Sound* 15(3):271–289.

[25] Schwartz, D., and M. Wright. 2000. "Extensions and Applications of the SDIF Sound Description Interchange Format." In *Proceedings of the International Computer Music Conference*, pp. 481–484.

[26] Stroppa, M. 19821986. *Traiettoria, for piano and computer-generated sounds*. Milan: Ricordi Edition.

[27] Stroppa, M. 1988. "Structure, Categorization, Generation, and Selection of Vertical Pitch Structures: A Musical Application in Computer-Assisted Composition." IRCAM Tech Report. Available on-line: http://articles.ircam.fr/textes/Stroppa88a/index.pdf

[28] Stroppa, M. 1989. "Musical Information Organisms: An Approach to Composition." *Contemporary Music Revue* 4:131–163.

[29] Stroppa, M. 2000. "High-Level Musical Control Paradigms for Digital Signal Processing." In *Proceedings of the International Conference on Digital Audio Effects – DAFx00.*

[30] Taube, H. 1991. "Common Music: A Music Composition Language in Common Lisp and CLOS." *Computer Music Journal* 15(2):21–32.

[31] Tenney, J. 1969. "Computer Music Experiences, 19611964." Electronic Music Reports #1. Utrecht: Institute of Sonology.