Visual Lisp/CLOS Programming in OpenMusic

Jean Bresson, Carlos Agon, Gérard Assayag

▶ **To cite this version:**

Jean Bresson, Carlos Agon, Gérard Assayag. Visual Lisp/CLOS Programming in OpenMusic. Higher-Order and Symbolic Computation, 2009, 22 (1), pp.81-111. 10.1007/s10990-009-9044-x . hal-00683472

HAL Id: hal-00683472
https://hal.science/hal-00683472

Submitted on 14 Mar 2014

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Visual Lisp/CLOS Programming in OpenMusic

Jean Bresson . Carlos Agon . Gérard Assayag
IRCAM - CNRS UMR STMS
Music Representation Research Group
Paris, France.

This is a preliminary version of the article *Visual Lisp/CLOS Programming in OpenMusic* published in *Higher Order and Symbolic Computation*, num. 22, vol. 1, 2009. The final publication is available at www.springerlink.com

Abstract

OpenMusic (OM) is a visual programming language developed on top of Common Lisp and CLOS, in which most of the functional and object-oriented programming concepts can be implemented and carried out graphically. Although this visual language was designed for musical applications, the focus in this paper is to describe and study OM as a complete general-purpose programming environment.

1 Introduction

Among its different historical fields, a particular approach in computer music research focused on the relations between musical and computer processes, in order to provide composers with means to develop musical formalisms and actualize musical ideas using the computer [6]. In this context, programming turned out to be a convenient solution, and computer-aided composition tools and software evolved toward specialized programming languages and environments [33, 50]. Due to their particular audience, the music programming languages share the necessity to deal and balance between efficiency, expressive power on the one hand, and user accessibility and creativity issues on the other hand. Functional programming languages, and particularly Lisp dialects, were often used for these purposes. The interpreted characteristic of Lisp allows an interactive use and an incremental development of compositional processes, and its relatively simple syntax, its expressivity and general capacity for the manipulation of symbolic structures allow to perform advanced operations on symbolic musical material [22]. Moreover, the mechanisms of abstractions and application of functional languages emphasize a symmetrical focus on the static musical data and the related generative processes, which proved to be pertinent from a compositional point of view. Other programming paradigms were also applied for music composition, like object-oriented programming [59], or declarative and constraint programming [43, 64]. These paradigms were also often integrated or implemented within Lisp environments. Formes [63] was one of the early computer-aided composition projects carried out at IRCAM in the 80s; It was at the same time a functional programming language based on Lisp, and one of the first object-oriented systems. Other Lisp-based music languages and projects include for instance Arctic [20] and Nyquist [21] environments, or Common Music [67]. SuperCollider [51] is another language for sound synthesis and music composition combining object-oriented and functional programming aspects in a C-like syntax.

Among these music programming environments, visual programming languages (VPLs) constituted a significant step forward. Theoretically, one can talk about visual programming when a language proposes a semantics of more than one dimension, that is, when it does not syntactically consist of purely linear text but when other visual elements are part of this semantics [13]. VPLs therefore let users develop programs by editing or manipulating such visual elements (symbols, shapes, pictures and their relative layouts or connections) according to a spatial/visual grammar. It is assumed that this graphical dimension is closer to mental representations [30, 41] and can make programming and using computer resources more productive and more accessible. If this idea can be contested (see for instance [71, 72] or [29] for studies on the impact of visual program representations and comparisons between visual and textual programs), it unquestionably fits certain domains and certain user communities (like music composers) willing to use computers to design complex processes but generally not particularly attracted or skilled in traditional text programming. As mentioned in [9], "textual programming languages have proven to be rather difficult for many creative and intelligent people to learn to use efficiently". Many studies and authors would indeed argue in the sense that imagery and graphical representations are privileged elements of the creative thought. VPLs allow to limit the possibility of syntactic errors while programming, but also contribute to a more interactive approach with respect to the relations between the user and the programs. In the case of music composition, this integration of programming, calculus and composition improves a great deal the potential of the computer tools. While keeping providing the programming languages' expressive power, they make it possible for composers to easily and progressively build and get into the complexity of the programs.

OpenMusic (OM) is a computer-aided composition environment developed at IRCAM, designed on top of Common Lisp/CLOS as a complete visual programming language [1, 7]. In the remainder of this article we will however leave the musical aspects aside in order to present OM as a general language and detail its main visual programming features and characteristics. After a brief survey of related works about VPLs (Section 2), we first expose the general approach and focus in the design of the OM language (Section 3). Then we explicit the basics of its syntax and semantics and show how functional expressions are represented, edited and evaluated (Section 4). In Section 5 we comment the integration of visual programming with textual Lisp programming, and then we detail the visual implementation of advanced programming techniques and constructs like functional abstraction and application, recursion (Section 6), higher-order functions (Section 7), and iterations (Section 8). Object-oriented programming is also supported in the language as a visual implementation of CLOS (Common Lisp Object System), and is addressed in Section 9. Some aspects of the metaobject protocol used in the OM language implementation are then introduced (Section 10), as well as the notion of persistence provided by the environment (Section 11). Before to conclude we finally expose some general remarks and analysis of the practical end use of OM by music composers (Section 12).

2 Visual Programming Languages – Related Works

VPLs can be classified in different categories according to their general program representation paradigm [15]. Such categories include icon-based languages (where basically programs are represented with iconic items, and semantics is given by their spatial relationships, e.g. [53]), form-based languages (e.g. spreadsheets like Form/3 [16] or Microsoft Excel), and diagram (or graph-based) languages. OM is part of the diagram language category. We shall therefore principally focus on this category in the remainder of this section.

The syntax of graph-based VPLs generally consists of processing modules (boxes) connected to each other by means of arcs which represent data flowing from a module to another. Diagrams provide a straightforward and intuitive representation of the processes being created, and are particularly adapted to the representation of functional expressions. A significative reference of diagram VPLs is Prograph [18], a general-purpose functional visual language allowing to define data-flow graphs using powerful visual constructs such as iterations, conditionals, or procedural abstraction, embedded in an object-oriented framework [19]. Other works on diagrammatic languages exist that present some comparable characteristics, such as the Pict and SunPict languages [27, 28], Show and Tell [39] and its different extensions (e.g. ESTL [56]), ProtoHyperFlow (PHF) [25], NUT [69] and Vista [65] for their object-oriented paradigm, or LabVIEW and

its underlying language "G" [70] which is probably the most successful commercially available VPL to date. A number of VPLs are explicitly dedicated to specific domains, such as experimental data and signal processing in DataVis [34] or in HP VEE [32], or image processing in the Khoros system and its embedded visual programming environment Cantata [73, 62]. Much like the Prograph model, all of these VPLs mostly relate to a data- or event-driven data flow model. In this model, the modules or functions "fire" data to downstream graph components when their parameters are supplied.

VPL [42] is another visual language dedicated to image processing which, at the contrary, has a demand-driven computation model. In demand-driven computation, execution starts from a downstream component and recursively triggers the computation of the connected upstream boxes needed to supply its parameters. This model is roughly similar to the way a functional expression is evaluated in an interpretive language. It is also the one which is implemented in OM.

Viz [36] is another early visual language which worth being cited here for it was specifically defined to visually represent functional semantics. Although our focus is on the field of diagram representations, we shall also mention works on functional visual programs carried out in "icon-based" languages like VisaVis [60], or some interesting attempts toward radically alternative program representations such as the Cube's 3-D representations [55], or the VIPR's nested rings representation for imperative programs [17]. Several interesting surveys and classification papers provide complete and structured overviews of the existing visual programming languages [54, 35, 9]. Visual programing languages and graphical program representation are therefore not recent or unexplored research fields, however few of the past research works and prototypes have actually been widely adopted, or at least made effectively available and still running on modern operating systems.

In the computer music domain, the Max system [61] was a pioneering visual graph-based language for the design and specification of real-time event and signal processing, from which are (at least conceptually) derived many of the environments currently in use for contemporary music creation. Reactive data-driven computation makes these environments particularly suited and oriented toward real-time interactive applications and live performance. On the other hand, PatchWork (Laurson *et al.* [44]) was another visual programming language dedicated to music composition and providing a graphical interface on Lisp, hence closer to pure functional programming, integrating specific high-level musical functions and data structures. PatchWork is actually the direct predecessor of OM. In contrast to real-time environments where the programs execute continuously from input streams (signals, events, metronomes, etc.), these languages execute programs on demand and provide a discrete synchronous control flow. This independence from real-time allows the generation of complex data, spread and structured over time and over other musical dimensions, which makes them well-adapted to formal compositional experimentation. Elody [57] is another example of a visual functional programming environment dedicated to music composition, which had the particularity to be an "icon-based" language allowing to perform abstractions and applications on musical data/function by simple drag and drop operations [48].

3 Elements of the Language Design

The historical development of computer-aided composition and the design of the PatchWork/OM languages imply various aspects broadly related to the emergence of graphical user interfaces in the 80s/90s. On the one hand, as stated above, the idea of visual programming was an attractive alternative for musicians to replace the traditional textual languages used in most computer music composition systems. As most of these systems where derived from Lisp, building an visual interface on this particular language seemed natural, and would allow to integrate and reuse underlying knowledge and tools developed in the past. On the other hand lay the idea of integrating "active components" in programs, that is, allowing users to build interactive models in an experimental framework combining algorithmic processes, data visualization and notation, and enabling the interaction of these different aspects and materials of a musical project.

In the mid 90s, two projects succeeded to PatchWork in different institutions: PWGL [45] and OM (Agon, Assayag *et al.* [1, 7]). Among the number of interface design issues and specificities resulting from these two separated branches, we can roughly underline that PWGL emphasizes particular aspects such as advanced

graphics and music notation or sound synthesis features embedded in the Lisp visual interface [46, 40], while OM would rather put forward a generic and modular visual programming framework. Specific features such as musical notation, time representation or sound synthesis are also present in OM, and have been discussed in previous articles [7, 2, 10]; We shall however ignore them in the present context in order to focus of the visual programming language itself.

The design of a complete programming language requires a minimal set of fundamental features which where indeed partially lacking in PatchWork. As most of the modern languages, PatchWork already provided procedural abstraction features in its visual programming framework. Abstraction is one of the most essential issues in programming languages, for it allows to create structured and scalable programs, easier to conceptualize, develop and maintain. Other important features were however missing, like higher-order programming, control structures such as conditionals and iterations, or recursion. Common Lisp includes all of these features through higher-order functions, anonymous and dynamically declared lambda functions, special forms and macros such as *if*, *loop*, and the like [66]. The corresponding constructs have therefore been implemented graphically in the OM language. They will be detailed successively in the following sections. In order to ensure maximum expressivity, we will see that they can also be completed or associated to textual Lisp programming when necessary, or when it appears to be more convenient.

Although it internally used some object-oriented aspects provided by CLOS, PatchWork did not provide either the users with any access to object-oriented programming, which is another important part of the OM visual language which will be addressed further on. In addition to standard object-oriented features like class and method definitions, multiple inheritance and polymorphism, several essential aspects specific to CLOS are implemented, such as the notion of generic functions (collection of methods specialized for different arguments), the multiple dispatch interface (specialization of the methods upon any or all of their arguments), or the standard method combinations [26].

4 Visual Programs in OM

OM falls into the category of the diagram, or "graph-based" VPLs. The diagrams created in the visual language are oriented graphs that correspond to functional expressions, ultimately translated to Lisp and evaluated. OM is an interpretive language, which means that programs can be created, modified and executed dynamically.

4.1 Basics of the Language Syntax and Semantics

Most visual programming in OM is done in *patch* editors. A patch is the general representation of a functional expression; it is constituted by boxes linked to each other via graphical arcs called *connections*. Figure 1 shows a patch implementing simple arithmetic operations.

Each box is associated to (or semantically *refers* to) an underlying element of the language (e.g. a function, a class, a patch[1]). In the example of Figure 1, the boxes refer to functions (+ and ×) or to constant values (3, 6 and 100).

A box is made of a kernel frame and a number of inlets and outlets, depending on the referred element. Inlets are visible at the top of the boxes, and outlets at the bottom. The inlets of a function box, for instance, correspond to the function arguments, and its outlets correspond to the (possibly multiple) returned values. When creating a program, the user has the possibility to access the characteristics of the boxes through documentation windows, and can know about

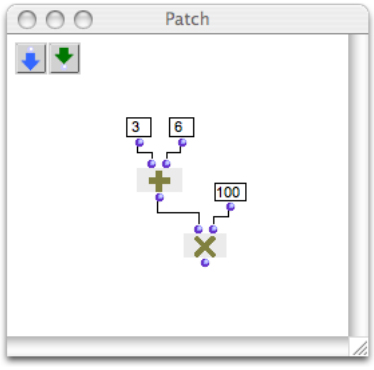

FIGURE 1: *A patch implementing the operation* $(3 + 6) \times 100$.

[1]Patches in OM are themselves part of the visual language primitives.

the individual assignment and possible values of each of their inputs and outputs thanks to a tooltips system (see Figure 2).

Semantically, each box corresponds to a functional application: Upon *evaluation*, it triggers a call to its reference (i.e. a function call, an object instantiation, and so on). The connections between boxes describe how data flows from one to another: They define the functional composition of the programs. By connecting an inlet of a box to an outlet of another box, a functional argument of the first box is set to be the result of the second one, and so forth. An automatic checking prevents the user to make syntactic errors such as creating cycles or connecting various different values to a same box input. Therefore, a set of connected boxes constitutes a consistent oriented acyclic graph that corresponds to a well-formed s-expression. The semantics of a patch is given by a set of rules that transform the visual expressions into Lisp expressions to be evaluated. The patch in Figure 1 corresponds to the following expression:

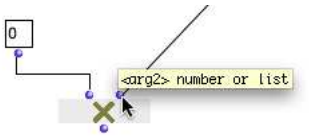

FIGURE 2: *Accessing information about the function arguments when connecting and manipulating a box.*

```
(* (+ 3 6) 100)
```

Each box has a *reference* and a *value*. At evaluation, the application of the box's reference computes the value.[2] The function responsible for this is a generic-function called *omng-box-value*, which is specialized for the different types of boxes. In the following code sample, *OMBoxCall* is a box that refers to a function:

```
; Eval the output indexed by 'numout' for the box 'self'
(defmethod omng-box-value ((self OMBoxCall) &optional (numout 0))
   (let ((args (mapcar #'omng-box-value (inputs self))))
      (nth numout (multiple-value-list (apply (reference self) args)))
   ))
```

During the evaluation of a box, the *value* of each of its inputs is to be determined before the actual function call. Inputs are formally defined by a *value* and a *connection*. The input *connection* is either null (if the input has no incoming connection) or a list (*box n*) indicating the connected box and the index of its connected ouput. If the input is connected, its value is therefore substituted by the *n*th output value of the connected box. Hence, the box inputs are key elements in the flow control: They transfer the evaluation to other boxes and return the results to their own box call according to the functional composition defined by the patch connections. The *omng-box-value* method for an input is defined as follows :

```
; Returns the value corresponding to a box input.
(defmethod omng-box-value ((self Input) &optional (numout 0))
   (declare (ignore numout))
   (if (connection self)
      (omng-box-value (car (connection self)) (cadr (connection self)))
     (value self)))
```

Evaluating a box therefore amounts to evaluating the functional graph (or the equivalent expression) rooted from this box. A recursive sequence of evaluations occurs which reduces the s-expression according to the patch functional composition. In Figure 1 the evaluation of the box × starts this reduction process: the result of box 100 (i.e. the value itself) is multiplied to that of box +, and so forth. The final result is stored as the value of box ×, and printed in a Listener window.

[2]Note that Common Lisp functions can return multiple values using a primitive called *values*. The caller of a function returning multiple values receives the first value by default, or all the values if the call is embedded in specific forms or macros: In the examples provided here after, we use the macro *multiple-value-list*, which collects in a list the possible multiple values returned by a function. In case of multiple return values, the corresponding OM box can present several outputs.

As we can see in this simple example, the computation model of OM is demand-driven: Depending on the purpose of the program, the evaluated box (i.e. the "root" of the evaluation graph) may be an object factory (see Section 4.2) or a simple functional call which result is printed in the Listener. Notice that the demand-driven model allows the program to be partially evaluated at each particular box in the graph. This is an important characteristic of the programming environment since it allows to progressively check the program correctness and debug its possible errors [30]. Various branch sharing common parts (or completely independent) can also coexist in a same patch, which makes it possible to test comparatively and locally some possible alternative or related processes.

4.2 Box Types

Various types of boxes refer to the different functional components of a program. Among the main box types are simple value boxes, function boxes, factory boxes, or patch boxes. Boxes can be added to a patch editor either by choosing the corresponding reference (functions or objects) in the window menus or in a specific "Library" window, or directly typing the name of the referred component (e.g. the function or class name) in the patch editor.

The function boxes refer to Lisp functions or to OM functions (we call OM functions particular types of functions designed in OM and including specific visual features such as icons, documentation, etc.) These reference functions can be built-in functions included in the running Lisp or user-defined functions created or loaded dynamically while using OM. The box + of Figure 1 is an example of a simple function box.

The patch boxes represent patches (i.e. programs) inside other patches. They are thus used to perform procedural – or more exactly, functional – abstraction (see Section 6).

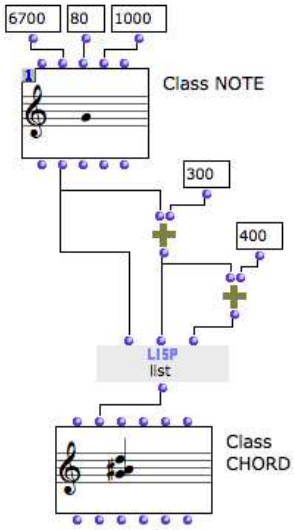

FIGURE 3: *The use of factory boxes in OM. A note object is instantiated from integer values corresponding to its pitch, velocity and duration. The pitch is used and processed in order to create a list of three values, in turn used to instantiate a chord object. In the OM musical objects convention, 100 corresponds to 1 half-tone. Adding 300 then 700 to a pitch amounts to adding a minor third and a fifth to the note for creating the chord.*

The factory boxes refer to classes and represent calls generating instances of these classes (equivalent to the CLOS *make-instance*). The inputs are the arguments of the constructor for this class, corresponding to its different slots (the class *initargs* in CLOS), and the outputs allow to access the values of these slots for the current (or last created) instance (*value* of the box). A special input/output in the factory boxes (the first from left) allows respectively to create (by copying or converting an other instance) or to access the instance itself as a value. Figure 3 shows an example of a patch using factory boxes. At the top of the figure is visible a *note* class factory. The inlets/outlets represent the note itself (first in/outlet from left) and the different slots of the class *note* (pitch, velocity, duration, etc.) As for function boxes, the slots names and documentation are accessible through the tooltips of the corresponding in/outlets or in the box documentation window. Depending on the class, the current factory box value can be displayed as text (values of the different slots) or with more specific representations (e.g. scores).

Figure 3 corresponds to the following Lisp expression:

```
(let ((note (make-instance 'note :pitch 6700
                                 :vel 80
                                 :dur 1000)))
    (make-instance 'chord :pitches (list (pitch note)
                                         (+ (pitch note) 300)
                                         (+ (+ (pitch note) 300) 400))))
```

Note that some boxes in Figure 3 have multiple connections (the *note* factory box at the top and one of the + function boxes). Therefore, they shall be called, and recursively call their upstream connected components several time during an evaluation process. This explains for instance that the expression (+ (pitch note) 300) is repeated twice in the corresponding Lisp code. The *note* instance, however, is created only once (and embedded in the *let* statement in the corresponding Lisp code): This is due to a special evaluation mode which will be discussed in Section 4.4.

4.3 Boxes and Containers

Most of the OM language elements (including the patches themselves) can be viewed either as boxes, i.e. compacted in a simple frame, or as containers, i.e. expanded in an individual window. A box can therefore be "opened", generally by double-clicking it, so that its contents can be modified in a dedicated editor (e.g. a patch editor, a generic function editor, a class editor, etc.) Schematically, the visual programming components are represented by a box in their container, and are internally accessible and modifiable thanks to an editor in which their internal elements are in turn represented by boxes and organized following specific syntactic rules.

The Lisp functions however (i.e. the primitive functions of the language or the predefined functions of the OM environment) are not defined graphically: They can be used in the form of boxes in other program containers but cannot be opened and modified in a graphical editor.

The factory boxes also deserve specific comments. Although OM classes can be edited visually (see Section 9 on object-oriented programming), a factory box editor will rather allow to visualize and edit the *value* of the box, i.e. the current (or last) instance it produced. The predefined object classes provided in OM (e.g. musical objects, curves, sound files, etc.) generally have specific editors that make it possible to edit the box value with appropriate interfaces (e.g. score editors, curve editors, etc.) The factory boxes therefore enable editing and modifying the data transmitted and processed in a program, and constitute privileged entry-points for the user to perform interventions at the various steps of the calculus [2].

4.4 Box State and Evaluation Modes

Boxes have a reference and a value, but also the ability to decide, depending on their *state*, when to call their reference and how to compute and return their value (which we also call its *evaluation mode*). The complete formal definition of a box actually corresponds to a triplet {*reference*, *value*, *state*}.

Keyboard and mouse shortcuts allow to change the state of a box and set it to several evaluation modes. A little icon informing about special states then appears at the top-left corner of the box.

When *state* is "normal", the box performs a reference call and the value is updated at each direct or indirect recursive evaluation, as described in Section 4.1. Two other box states or evaluation modes are of a particular interest.

In the *locked* state, the value is computed once and and kept unchanged for all of the following evaluations until the box is unlocked. The further graph evaluations are stopped when reaching a locked and pre-valued box, and hence do not trigger the reference call, nor that of the possible upstream boxes.[3] A locked box therefore semantically correspond to a kind of constant function inserted in the evaluation graph. Note that manual edition of the factory boxes affect their value. They are therefore automatically set to the *locked* state when modified in order to avoid the possible manual modifications to be overridden by further box evaluations.

Another box state is called *eval-once*. To understand the corresponding evaluation mode we must first differentiate the "general" evaluation triggered by the user and the "indirect" (or "recursive") box evaluation that occurs when a connected box requires the value of this box. When the box state is *eval-once*, it will evaluate normally at its first call during a general evaluation and then behave as a locked box during the remainder of this evaluation. This behaviour allows to prevent useless computations when a box has to be evaluated several times in a same general evaluation (i.e. when its outputs are connected to various other

[3]Here "upstream" is meant in the sense of the data flow.

boxes in the graph, or when it is upstream an iterative control). In Figure 3, the *note* factory box is in state *eval-once*, as indicates the little "1" icon at the top-left of the box. As we can see in the corresponding Lisp expression, the *eval-once* mode makes the box simulate a Lisp *let* statement and behave as a local variable declaration. In contrast, the + box is not in any specific state although it has two downstream connections, which explains the repeated code in the Lisp expression.

Note that in case of nondeterministic processes, the *eval-once* mode also allows to ensure the consistency of the results by returning the same output value to its different successive callers.

4.5 Documentation and Secondary Notation

Each function or class can be assigned a documentation. The OM classes and functions can moreover be set particular documentation and extra information for each of their function arguments, class slots, etc. (see Section 10). From a visual program, it is possible to access the documentation and information for a given box through various structured representations such as a tooltips or documentation windows. It is also possible to create inline tutorials or example patches to be assigned to the different functions and classes.

Various means such as text comments, colors or background pictures, also allow to enhance secondary notation [58] and convey extra information in a visual program.

5 Relation to Text Programming

Supporting text programming complementarily to the visual aspects is an important issue for a visual programming language. Using text programming is indeed often a good solution to cope with the complexity of a program, because of the compactness of the textual representation, or because sometimes things can just be more easily or more efficiently expressed with text than in a visual program. In real-size projects, for scalability issues or for general convenience, it is often necessary to have some components designed or reused under the form of textual functions and components.

OM is not a "naturally" visual language, in the sense that it is actually an augmented visual implementation of a Lisp environment. A set of Lisp editing tools and a traditional Lisp interpreter are accessible and smoothly integrated in the visual programming framework. Any function, method, or class defined in Lisp, in OM or independently from the visual environment, is immediately accessible and can be included in the OM patches (see Figure 4).

A documented protocol allows to easily create and load external user libraries in OM. These libraries generally consist in specific-purpose collections of functions that can be loaded dynamically and used transparently in the OM patches.

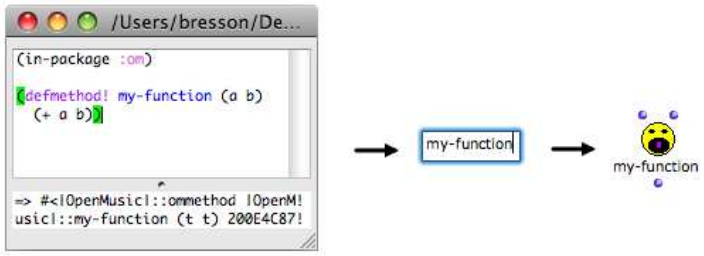

FIGURE 4: *Definition of a method in Lisp and creation of the corresponding box.*

When a box's reference is a Lisp function or class, and when the source code of this function or class is available in OM or in the user's code, a keyboard shortcut allows to find and open it in a Lisp editor. The box's reference (function or class) can therefore be modified and dynamically redefined. Conversely, it is possible to use in the Lisp-written code some functions or classes defined graphically, which makes OM falling into the category of Heterogeneous VPLs (HVPLs) as defined in [24]: "both types [of programming]

can be arbitrarily mixed and can even operate on the same structures. The programmer is entirely free to choose whatever fits best and to find his personal balance between both modes."

Despite this tight integration, however, there is still a important formal distinction between "graphical" programming components (patches, functions or classes defined in the visual language) and "traditional" Lisp components defined textually. A function defined textually can not be represented and edited as a visual program, and a visual program, function, or class can not be directly displayed and edited as text. The latter conversion could be envisaged with no major problem, though, but at the loss of all graphical information (principally the patch layout and box settings). This makes the text-to-graphical conversion much more difficult for it would require the automatic computation of the graph layout and box positions, but also an exhaustive graphical matching and interpretation for every Lisp statements and constructs.

6 Functional Abstraction

Functional abstraction basically consists in making some elements of an expression become variables. Inputs and outputs, also represented by boxes, can be introduced in an OM patch to represent these variables in the graph.

Starting from the example in Figure 1, for instance, it is possible to create the function $f(x, y) = (x + 6) \times y$ by adding two inputs and one output connected to the program as shows *patch1* in Figure 5 (a).

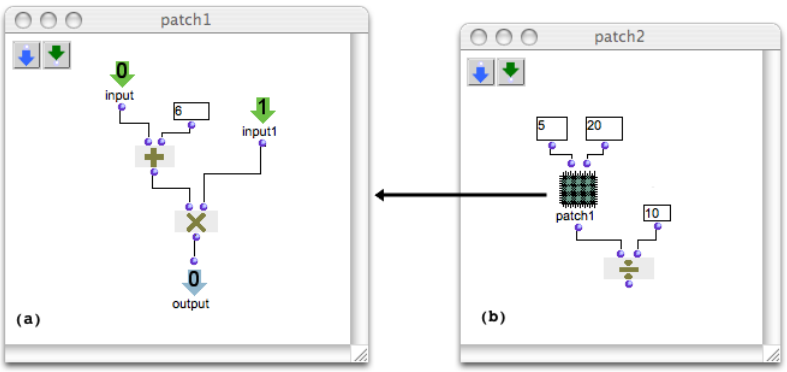

FIGURE 5: *(a) Definition and (b) application of a function. Abstraction is carried out by making variable some elements of the program.*

Hence *patch1* then corresponds to a function definition. The corresponding Lisp expression would be:

```
(lambda (x y) (* (+ x 6) y))
```

As a function definition, it can also be expressed as follows:

```
(defun patch1 (x y) (* (+ x 6) y))
```

Input boxes are identified by an index and completed with a name, a documentation string and a default value. This information correspond respectively to the index and name of the patch box inlets, their possible help string and default value in the patch application (see next section).

6.1 Functional Application: Patch Boxes

The function defined in a patch can be included in another patch, where the data can be supplied to the abstraction variables (functional application). In Figure 5 (b), *patch1* is included in *patch2* as a function with 2 arguments and 1 output value. In this example, we see a new type of box (the box labelled *patch1*), which refers to a patch. Its inlets and outlet correspond to the input and output boxes in the patch, and its evaluation would correspond in this case to the Lisp call:

```
(patch1 5 20)
```

The complete graph in *patch2* therefore in turn corresponds to the expression:

```
(/ (patch1 5 20) 10)
```

Within a patch editor, the program can therefore be modified and partially executed. From the outside, however, it is a box corresponding to a simple function which can be moved/copied and used in different contexts and purposes. The multiple occurrences of a patch box in other patches all refer to the same function.

In these abstraction/application mechanisms, the graphical expressions are converted into real Lisp functions. A process internally carries out this conversion by a recursive call to a code-generating method on the patch functional graph. During this process (called *compile-patch*) each box generates the Lisp code corresponding to its reference, including the code recursively generated by the boxes it is connected to. For instance, a simple function box code generation looks like :

```
; Generates the Lisp code for a function box at output 'numout'
(defmethod gen-code ((self OMBoxCall) &optional (numout 0))
   (if (zerop numout)
      `(,(reference self) ,@(mapcar #'gen-code (inputs self)))
     `(nth ,numout (multiple-value-list ...))))
```

In order to generate the full patch expression, the code-generation process must start with the patch output boxes. As in the case of evaluations, the box inputs allow to follow the functional composition of the patch. The *gen-code* method of an input will either report the call to the box it is connected to or, if the input is not connected, return its own value textually. Internal locked boxes will also directly return Lisp expressions corresponding to their current value, and input boxes will finally generate the name of the corresponding arguments in the function being defined. The generated Lisp expression is then evaluated as a function definition attached to the patch. Here is a simplified version of the *compile-patch* function:[4]

```
; Generates the Lisp function for patch 'self'
(defmethod compile-patch ((self OMPatch))
   (let* ((out-boxes (output-boxes self))
          (in-boxes (input-boxes self)))
      (eval `(defun ,(patch-function self) (,@(mapcar #'name in-boxes))
                  (values ,@(mapcar #'gen-code out-boxes)))
      (setf (compiled?  self) t)
   ))
```

The patch keeps track of user modifications, so this process can take place only when needed, in case of modification and in particular occasions (e.g. before a call to this patch or when the patch editor window is closed by the user). The evaluation of a patch box (class *OMBoxPatch*) then consists in the application of its attached function to the values connected to its inputs:

```
; Eval the output indexed by 'numout' for the box 'self'
(defmethod omng-box-value ((self OMBoxPatch) &optional (numout 0))
   (let ((thepatch (reference self))
         (args (mapcar #'omng-box-value (inputs self))))
      (unless (compiled?  thepatch)
         (compile-patch thepatch))
      (nth numout (multiple-value-list (apply (patch-function thepatch) args)))
   ))
```

[4]The actual process is in fact a bit more complicated and requires further operations for handling local variable declarations (i.e. boxes in *eval-once* evaluation modes), unique names for function name and arguments, and other particular behaviours.

6.2 Local Functions

Complementarily to the abstraction mechanism detailed above, it is possible to create sub-programs (or sub-patches) which are local functions, defined only within the local scope of a patch. Local functions can be created directly in the patch editors, or from already existing patch boxes (by creating a local copy of the patch). These sub-patches are graphically differentiated with the color of their referring boxes. They allow to create self-contained programs with fewer dependencies, at the loss of modularity: While all boxes referring to an abstraction pointed to a unique patch, local functions are duplicates edited independently in each of their occurrences.

For instance in the example of Figure 5 *patch1* is an abstraction defined in the environment, which would correspond to the following expressions:

```
; patch1
(defun patch1 (x y) (* (+ x 6) y))

; patch2
(/ (patch1 5 20) 10)
```

With a local function, we have the possibility to obtain something similar to:

```
(flet ((patch1 (x y) (* (+ x 6) y)))
  (/ (patch1 5 20) 10)))
```

In this case, *patch1* does not exist outside *patch2*.

6.3 Recursion

A patch box can be dragged in its own editor window, which corresponds to using an abstraction in its own definition, and implements the notion of recursion. Figure 6 shows a patch corresponding to the recursive function "factorial":

```
(defun factorial (x)
   (if (= x 1) x
      (* x (factorial (- x 1)))))
```

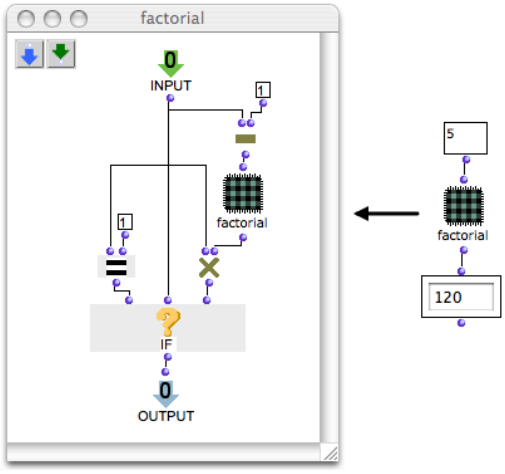

FIGURE 6: *The recursive function "factorial" in OM.*

6.4 Textual Abstractions

Another possible way to integrate text and visual programming is to create *lisp*-patches. *Lisp*-patches are edited textually as lambda expressions and used as abstractions in other patches. The lambda expression is just evaluated and the resulting function is attached to the patch, which referee boxes' inlets and outlets are updated accordingly (see Figure 7).

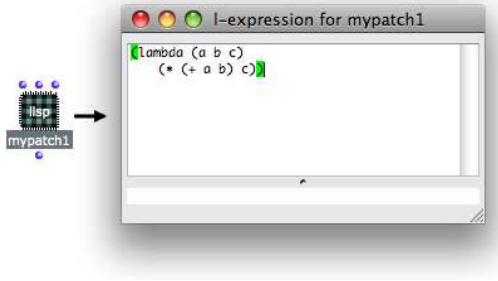

FIGURE 7: *Editing a patch as a textual lambda expression.*

7 Higher-Order Programming

In functional languages like Lisp programs and data are generally both considered as "first-class" objects. A Lisp function can thus be considered as data and inspected or constructed in the calculus. This allows for the creation of higher-order functions, i.e. functions accepting other functions as arguments, or producing functions as output values.

OM boxes (i.e. function boxes, but also patch boxes or factory boxes) can have their state set to another mode called "lambda", so as to return their reference call as a functional object (an anonymous function) instead of the result of this reference's functional application. When a patch box is in mode *lambda*, a small "lambda" icon is displayed on it. Concretely the *omng-box-value* method of the box will return a lambda expression instead of evaluating the reference call as it is the case in the "normal" state. A box like +, for instance, would return a function equivalent to `(lambda (x y) (+ x y))`; the patch *patch1* (Figure 5) would return a function equivalent to `(lambda (x y) (* (+ x 6) y))`, and so forth. The resulting function can then be used and eventually called as such in the continuation of the program execution. Figure 8 shows examples of the use of *patch1* with the lambda mode.

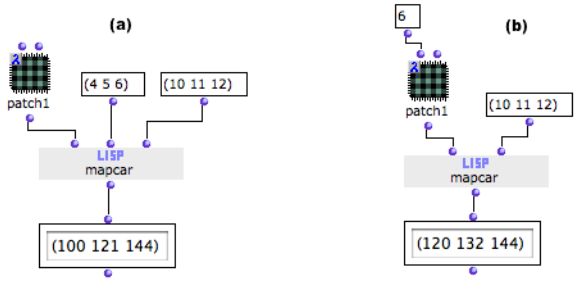

FIGURE 8: *(a) Creation of a lambda form in a visual program. The patch box patch1 is in mode lambda and returns a function, called using the mapcar list iteration with arguments '(4 5 6) and '(10 11 12). (b) Currying: the previous function is converted to a function of one single argument by explicitly setting one of the input values in the lambda form.*

Equivalent Lisp expressions for the two examples in Figure 8 are:

```
(mapcar #'(lambda (x y) (* (+ x 6) y)) '(4 5 6) '(10 11 12))
>> (100 121 144)

(mapcar #'(lambda (y) (let ((x 6)) (* (+ x 6) y))) '(10 11 12))
>> (120 132 144)
```

Figure 9 shows another example of the use of a lambda function as a test of equivalence between the elements in a list.

Currying (i.e. transformation of a function of n arguments into a function of $n-1$ arguments) can also be carried out using the *lambda* state, by connecting some of the inputs of the box. In this case the connected inputs are not considered as variables of the lambda form but evaluated and embedded in the lexical closure (see Figure 8 – b).

8 Control Structures

Various control structures commonly used in programming languages are available in the form of special boxes and editors in OM. Figure 10 shows an example of an *omloop*, which represents an iterative process (equivalent to a Lisp *loop* statement). The *omloop* box visible on the left is associated to a special patch editor in which the iteration process is defined.

Different *loop* iterators, counters and accumulators are available and visible at the top of the window in Figure 10. They are used and combined in the *omloop* editor in order to control the iteration. In this example the iteration is done on the list given as the input of the loop via the *list-loop* iterator (other available iterators include *while*, *for*, *on-list*). The box *eachtime* evaluates its incoming connection triggering a demand-driven execution at each iteration. It corresponds to the *do* statement in the Lisp *loop*. The *collect*

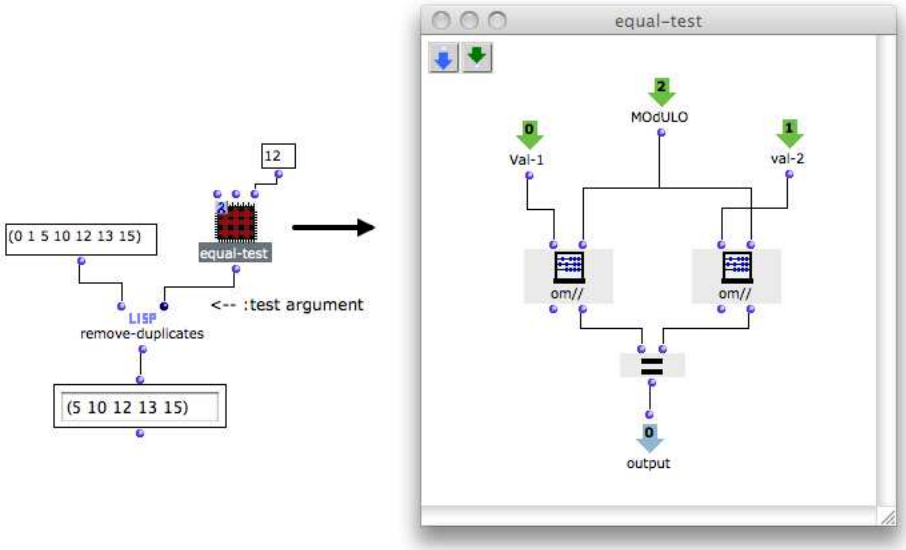

FIGURE 9: *Using a lambda function: the patch equal-test tests if two values are equal modulo 12, i.e. if the remainder of their division by 12 is the same. This patch is used as a test to remove elements in lists with remove-duplicates. In a musical application, this procedure amounts to removing octaves in a pitch list (one octave = 12 semitones).*

box simulates a temporary memory and acts as the *collect* statement in the Lisp *loop*. It collects elements in a list when evaluated from its first outlet, and returns the resulting list when evaluated from its second outlet (evaluation of the third outlet reinitializes the memory). Other similar accumulator tools are available, like *sum*, *max*, *min*, etc. Users can also define their own loop accumulation rules. The box *finally* is evaluated when iteration terminates and returns the result of the *omloop* (in this case, the content of the *collect* memory).

At each step of the iteration of Figure 10, hence for each element in the list, another control structure is used: the conditional *omif*, which corresponds to the Lisp *if* statement (also seen in the previous example on Figure 6). This conditional box provides flow control by evaluating its second input ("then") or its third input ("else") depending on the result of a condition connected to its first input. Here, the values from the list are incremented if they are inferior to a given threshold or returned as such if not. The successive results are collected in the list returned as the result of the iteration. This loop corresponds to the following Lisp function:

```
(lambda (list)
 (loop for x in list
   collect (if (>= x 5) x (+ x 5)))))
```

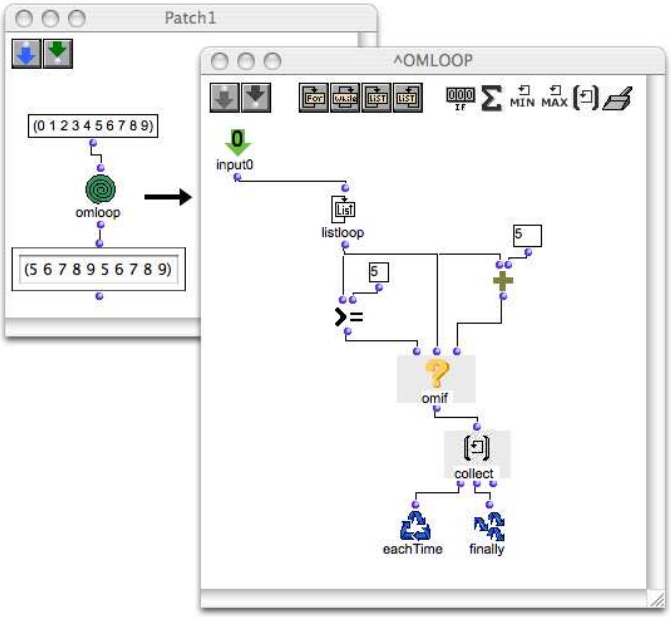

FIGURE 10: *omloop: iterative list processing: for each element in the list, collect element if* >= 5 *or* (*element* + 5) *if not.*

The *file-box* tool is another example of a visual iteration, performing the equivalent of an *omloop* within a *with-open-file* statement, i.e. with an input and/or output access to a file stream (see Figure 11).

The *file-box* in Figure 11 corresponds to the expression:

```
(lambda (path list)
   (with-open-file (s path)
      (loop for item in list do
         (write-line item s))))
```

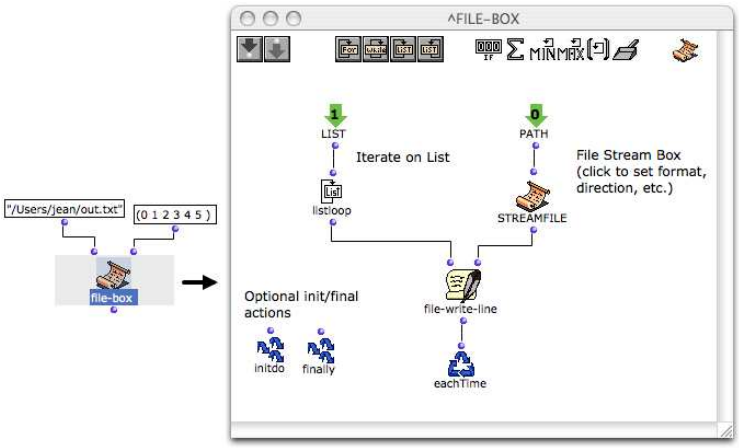

FIGURE 11: *filebox: i/o access on file streams in OM. The streamfile box represents the stream declaration, initialized with a pathname.*

9 Object-Oriented Programming

In addition to the functional programming tools presented above, OM includes a visual object-oriented programming framework built on top of CLOS [26].

9.1 Class Definitions

The OM environment contains several thematic packages, in which predefined or user-defined functions and classes are organized. A class tree editor for the different OM packages is accessible. Figure 12 shows the class tree of the *score* package. Classes can be created and modified graphically from this window. After selecting a name and an icon, the contents of a new class is specified using the class editor visible in Figure 13. Basically, class editing consists in adding/removing slots, setting their names, types, and default values. These operations are mostly done by typing text or graphically by drag and drop operations from the other classes or objects in the OM environment.

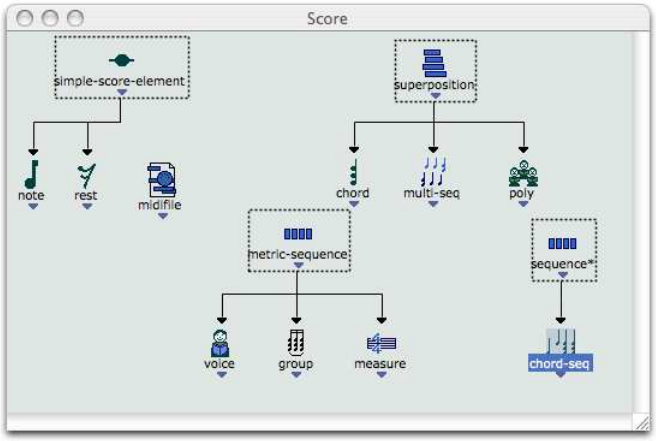

FIGURE 12: *Class tree editor of package score. Inheritance relationships are represented by arrow-shaped graphical connections. The dotted frames indicate alias boxes that refer to classes from other packages.*

Inheritance relationships can be set in the class tree editor (see Figure 12). Users cannot modify the OM predefined classes; however they can define new classes inheriting from them. As the class tree editor only contains and displays the classes of a given package, *class alias* boxes can be used to represent the classes from foreign packages required for creating such inheritance relationships (the dotted box frames in Figures 12 and 13).

The equivalent Lisp expression for Figure 13, automatically generated and evaluated in this situation, is:

```
(defclass!  myclass (chord)
    ((env :accessor env :initarg :env
          :type bpf :allocation :instance
          :initform (make-instance 'bpf))
     (val :accessor val :initarg :val
          :allocation :instance
          :type integer :initform 0))
     (:icon 212)
     (:documentation ""))
```

A class defined in OM can be dragged in a visual program where it is represented by a corresponding factory box (remember the *note* and *chord* factory boxes in Figure 3).

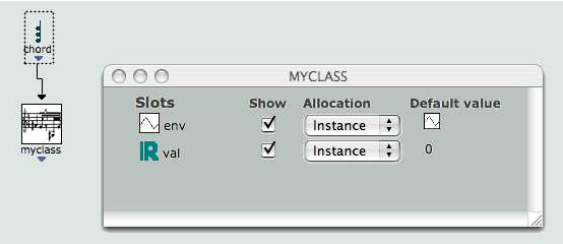

FIGURE 13: *Graphical definition of class myclass, extending the predefined OM class chord (in this example, chord is an alias, since the chord class is not in the user package). The class editor is open at the right of the figure: two additional slots are created: env and val. The icons next to the slot names represent their type.*

9.2 Generic Functions, Methods

The polymorphism of the generic functions in CLOS is also integrated in the OM visual language. Like OM classes, the OM generic functions and their different methods are organized and accessible in the different packages, from which they can be selected or edited.

Opening a generic function displays its internal methods in the generic function editor. Methods can be added, removed or edited from this window. Figure 14 (a) shows the editor for the generic function *om+*, which lists its four existing methods. Each method specializes the generic function for the different possible types of its arguments.[5] The generic function *om+* is actually a predefined OM function so its internal methods can not be edited graphically. It is possible, however, to add new methods to this function and specialize it for specific types of arguments.

Figure 14 also shows the editor of a new method being defined for the generic function *om+*. This editor looks like a patch editor but it has a fixed number of input boxes corresponding to the arguments of the generic function. These input boxes have the particularity to be *typed* inputs which will identify (i.e. specialize) this particular method. The specialization of the method arguments is done by dragging class boxes on the different input boxes. A typed input box then presents the required outlets that allow to access directly the different slots of the argument value (see for instance the first input of the method in Figure 14, specialized with the class *chord*).

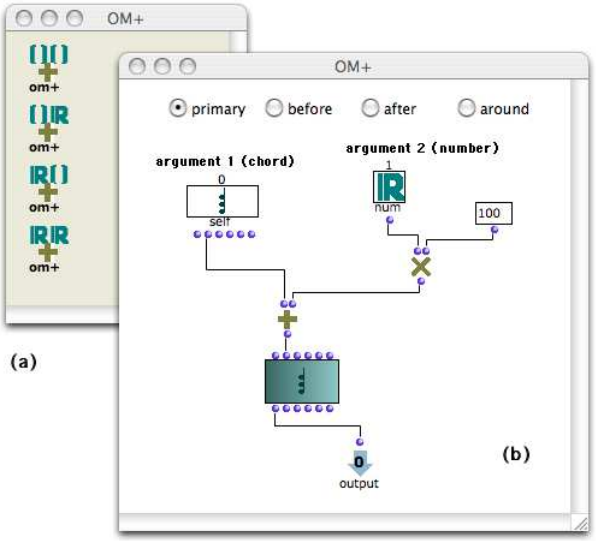

FIGURE 14: *Method definition. A new method specializes the generic function om+ for arguments of types chord and number. This method returns a chord which pitches are the addition of the pitches of the chord given as a first argument (self) + the second argument (a number) × 100 (100 means 1 half-tone for OM musical objects).*

[5]Remember that in CLOS, a generic function gathers a set of methods corresponding to its various possible specializations. The multiple dispatch system makes it possible to specialize methods upon any or all of their arguments [26].

The auxiliary methods *:before*, *:around* and *:after* provided by the CLOS standard method combination can also be created by selecting the corresponding option at the top of the method editor.

The method created in Figure 14 (b) corresponds to the following Lisp definition:

```
(defmethod!  om+ ((self chord) (num number))
   (make-instance 'chord
                  :pitches (om+ (pitches self) (om* num 100)))))
```

Among the other object-oriented programming features available in OM are also the possibility to define a specific processing function called at creating an instance of a class (corresponding to *initialize-instance* in CLOS), or to redefine graphically the accessor methods for its different slots. Figure 15 shows the visual redefinition of the *initialize-instance* method for the class defined in Figure 13.

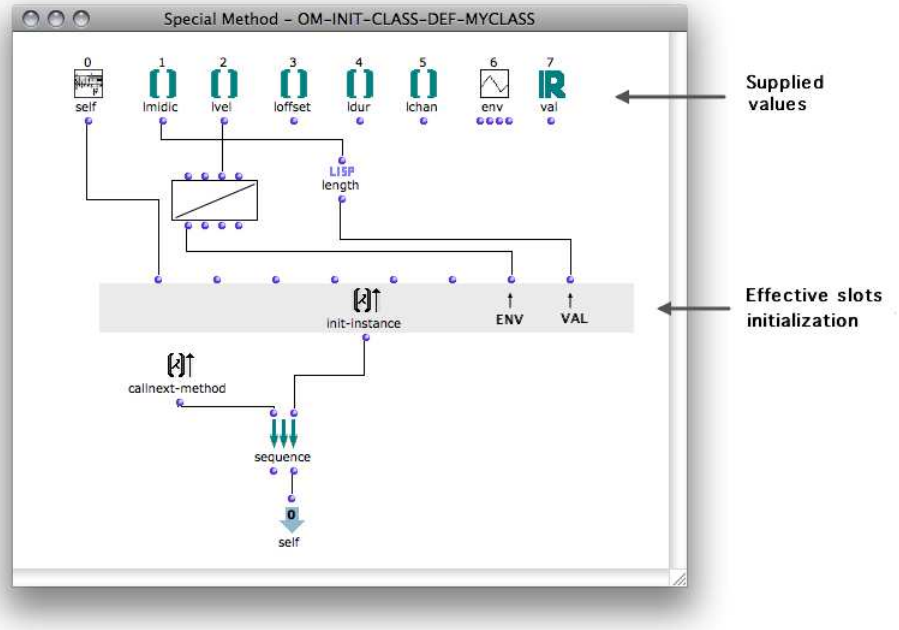

FIGURE 15: *Visual definition of the initialize-instance method for class myclass. The slot env (for "envelope") is initialized with a 2D curve describing the dynamics of the chord (myclass is a subclass of chord), and the slot val is initialized with the number of notes in the chord. Note the use of call-next-method, which calls the superclass' initialize-instance prior to the one defined in this method, as well as the flow control box sequence which, upon evaluation of its nth output, evaluates sequentially all of its inputs and then returns the nth result.*

10 The MetaObject Protocol

10.1 Elements of the Language Architecture

In CLOS, classes, functions, methods, and other element of the language are metaobject classes, which can be handled thanks to the MetaObject Protocol (MOP [37]). The MOP yields reflexive properties to the language: The elements of the programs, instances of metaobject classes, can be processed and modified at runtime by these same programs.

The OM language metaobjects (*OMClass*, *OMGenericFunction*, *OMMethod*, etc.) extend the standard CLOS metaobject classes, and have been set particular properties related to specific visual aspects of the

language (icons specification, documentation, persistence, behaviours in the graphical user interface, and so on). *OMClass*, for instance, is a subclass of *standard-class*. It has a slot called *icon* containing an icon ID converted to a picture when the class has to be displayed on screen:

```
(defclass omclass (standard-class)
    ((icon :initarg :icon :initform 0)
    ...))

>> #<standard-class omclass>
```

CLOS allows to determine the class of a metaobject (i.e. its *metaclass*): From there on, *OMClass* will be the general metaclass of all the OM objects and metaobjects. In other words, all classes in OM are defined as *OMClasses* instead of being *standard-classes*:

```
(defclass my-class ()
    ((slot1 :initarg :slot1 :accessor slot1)
     (slot2 :initarg :slot2 :accessor slot2))
   (:metaclass omclass))

>> #<omclass my-class>
```

The OM metaobject classes themselves are also instances of *OMClass*. In the following example, we see that *OMMethod* is both a subclass of *standard-method*, and an instance of *OMClass*.

```
(defclass OMMethod (OMBasicObject standard-method)
   ([...  OMMethod slots ...])
   (:metaclass omclass))

>> #<omclass ommethod>
```

Specific protocols are established for an easier instantiation of OM metaobjects (i.e. class/method/function definitions). The macros *defclass!* and *defmethod!* expand as calls to *defclass* and *defmethod* that specify the appropriate metaclass for the new metaobject and allow for the setting of its specific attributes. An *OMClass*, for instance, can be created directly as follows:

```
(defclass!  my-class2 () ()
   (:icon 21))

>> #<omclass my-class2>
```

The same principle applies for generic functions and methods. In the following example, the keywords *icon*, *initvals* and *indoc* allow to specify the attributes of the *OMMethod* instance which is created, that will be used to specify respectively an icon for the graphical representation of this method, and the default value and documentation for each of its arguments:

```
(defmethod!  my-method (arg1 arg2)
      :icon 123 :initvals '(0 0)
      :indocs '("argument1" "argument2")
   (+ arg1 arg2))

>> #<ommethod my-method>
```

This syntax is used in the OM or user code for defining functions and classes, and is also generated when they are defined graphically (see for instance the "equivalent" Lisp definitions in Section 9).

10.2 Visual MOP (VMOP)

In the current distributed release, there are no standard tools in OM for handling metaobjects graphically (i.e. creating new classes of classes or classes of functions, extending their behaviour, and so forth). This is possible in practice, though, and was part of the original design and implementation of the language as presented in [1]. A description of OM metaobject programming features (VMOP – Visual MetaObject Programming) and some applications are given in [3].

It is worthy of note that the boxes and other visual components which provide the visual language's graphical representation and user interaction framework are other basic elements of the visual language specification (called "visual metaobjects"). The visual metaobject classes and the visual features in general are therefore part of the language, which enables the actual reflexive aspects in OM. Interesting applications enabled by visual metaobject programming are for instance the possibility to modify the evaluation behaviour of some types of boxes (e.g. adding graphical indications on a box when it is computing or waiting for upstream boxes computation), or even to transform the language execution model (e.g. making the connection of the box triggering computations). Examples of such applications are also detailed in [1] and [3].

11 Persistence

11.1 Workspace and Persistent Objects

The main window of the OM environment is called a *workspace* and is similar to a classical OS desktop. In this workspace the user creates his/her programs (patches) and organizes them in directories. Figure 16 shows an OM workspace. Each icon represents a patch or a folder containing patches or sub-folders. From the workspace, patches can be open and edited as shown in the previous sections. They can also be dragged to an already open patch editor in order to be represented in it as a patch box, as shown in the example of Figure 5.

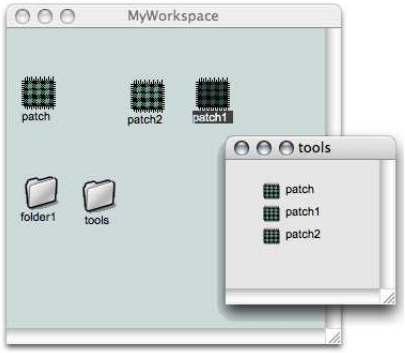

FIGURE 16: *A workspace in OM.*

The workspace organization reflects a real file architecture on the disk: patches are "persistent" objects. They are saved as Lisp expressions which evaluation recreate the original boxes and graphs, eventually displayed in the patch editors. Similarly, the user-defined metaobjects (classes or methods) are organized in packages and accessible via a specific package browser window. They can be edited from there or dragged to a patch editor in order to take part of a program being created. They are also saved as Lisp expressions in files, so that the user can save his whole programming framework contents and later reload his programs, classes and functions as in a traditional programming environment. Global variables are also implemented and stored as persistent objects: They can be created in the environment and referenced in visual programs as well.

These persistence and storage features allow significant code reuse possibilities, which is a major issue in visual programming language design [14]. The elements of the workspace (principally, the patches, that one could compare to atomic modules or functions) and of the packages architecture (respectively corresponding to the CLOS functions and objects) constitute the building blocks of an overall project or programming framework, or maybe, in terms related to computer-aided composition, of a compositional "modeling space" [49]. These functions and objects can be stored independently and used freely in one another, allowing an almost unlimited scalability and complexity of the programs.

11.2 How It Works

A generic function *omng-save* determines the self-generating Lisp expression for saving every type of object in the OM environment.

Atomic data types simply return their value. For compound data structures, however, similarly to what we have seen with the code generation of a visual program, a recursive process generates this expression from top-level structures down to their internal components. A simple example is for instance the saving method for a list, which is defined as follows:

```
; Generates the Lisp code for saving/reloading a list
(defmethod omng-save ((self list))
   `(list ,@(mapcar #'omng-save self)))
```

Following this principle, and with some simplification, the definition of *omng-save* for a standard class is as follows:

```
; Generates the Lisp code for saving/reloading instances of class 'C'
(defmethod omng-save ((self C))
   `(make-instance ',(class-name (class-of self))
                   ,@(mapcan #'(lambda (slot)
                         (list (slot-initarg slot)
                               (omng-save (slot-value self (slot-name slot)))))
                      (class-slots (class-of self))) ))
```

The instances of this class (or its subclasses) will therefore respond to *omng-save* with the following expression, where *value1*, *value2*, etc. initialize the slots with the current values of the instance attributes:

```
(make-instance 'C :slot1 value1 :slot2 value2 ...)
```

For the boxes and other top-level graphical components of a patch, the generated code must contain the internal *reference* (i.e. the class name, function name, sub-patch, and so forth) which will allow to recreate the right type of box, but also its graphical attributes (position, size, and the like), its *state* and its internal *value*, as well as that of each of the different inputs. The *omng-save* method for a box, here also with some simplifications, will therefore look like:

```
; Generates the Lisp code for saving/reloading a box
(defmethod omng-save ((self OMBox))
   `(make-new-box ,(name self)
                  ,(reference self)
                  ,(box-position self) ,(box-size self)
                  ,(omng-save (value self))
                  ,(state self)
                  ',(mapcar #'omng-save (inputs self)) ))
```

Finally (or first of all) a top-level persistent object (e.g., a patch, method, etc.) will simply write in a file its own generated expression, including the recursive *omng-save* call to its internal boxes.

12 Discussion: OM in Practice

It is hard to select and present typical applications of the use of OM since they could never really stand for representative examples of how and what such as visual programming language is to be used for. Nevertheless, the numerous practical applications and experiments gathered around OM shall let us draw some general observations. The following example patches (Figures 17, 18, 19) are extracted from recent pieces created in OM and reported in [12].

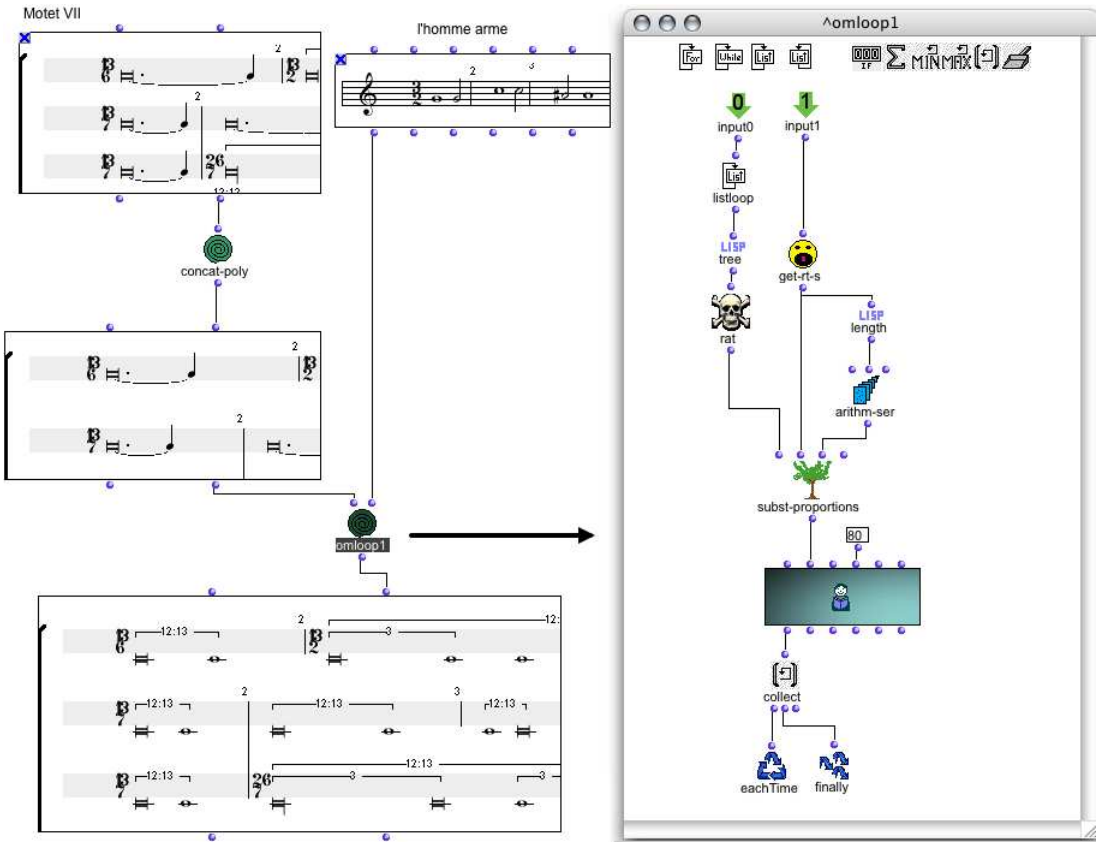

FIGURE 17: *Patch by Karim Haddad computing a rhythmic part in Livre Premier de Motets (2007) [31]. The omloop box editor is open at the right. Note: The input data boxes of this patch are in the locked state (musical object factories with a little "×" icon). These boxes are initial material of the program; They have been either filled or modified manually, or imported from other programs. Their contents will not be reinitialized at evaluation time.*

We shall first mention that the role and position that the visual programming and computer-aided composition environment in general plays in a given work or compositional approach can be very different from one composer to another. If some composers will use it for generating a whole piece in a program run (a kind of "algorithmic composition" approach), others will just consider the environment as an experimental framework where ideas and models are tested, but from which no actual musical material will be output and actually used in a piece. In between are the major part of OM users, who use the language, depending on their projects and purpose, to determine some specific musical data (e.g. rhythmic or harmonic structures), to generate abstract forms and material, or to perform operations and treatment of this musical material (transformation, automatic analysis, processing, etc.)

The textual vs. visual issue also deserves a word. In this respect, our experience showed that depending on the user and his/her goals, OM can be used either as a fully graphical language, where all is done via visual programs (from low-level processes to large-scale modular programming), or as an integration environment where Lisp-programmed tools and functions are just put together in a global project. Most of the time the reality is also between these two extreme cases; At least, the use of predefined musical functions and classes provided in OM position the standard user in an intermediate situation. In the examples provided here, we can see that some of the function boxes used by the composers are either Lisp primitives (e.g. *length* in Figure 17), predefined OM functions (e.g. *arithm-ser* in Figure 17, or *om-scale*, *flat* in Figure 18), or user-defined functions performing more specific operations loaded from libraries or written by the composer himself (e.g. *subst-proportions* in Figure 17, substitution of rhythmic patterns by others in a hierarchical rhythmic

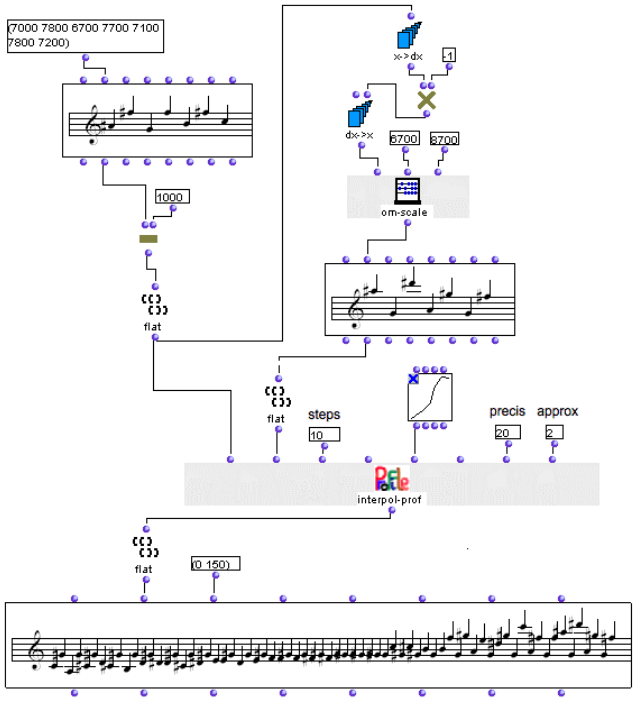

FIGURE 18: *Patch computing interpolations between a melodic profile and its own rescaled inversion, by Christopher Melen [52].*

structure, or *interpol-prof* in Figure 18, interpolation between melodic lines following specific profiles and constraints). A frequent situation is that the user first experiments with visual programs, then encapsulates relevant tools in Lisp modules as these tools get sufficient generality and maturity, and as he/she gets used with Lisp programming.

Finally we shall end with some comments about the use of the different programming paradigms available in OM. Functional programming seems to be "naturally" integrated by music composers, as well as the data flow graph model in general (Figure 19 is an example of the use of "lambda patch" boxes and higher-order programming). Although they were not addressed in the present paper, constraint programming tools also regularly find concrete and interesting applications. Harmonic, rhythmic or polyphonic structures have all been subjected to numerous successful formalization in the form of constraint satisfaction problems. Many of such applications are reported by OM users in [5, 12].

Object-oriented programming, however, has found fewer applications. If this programming approach is continuously used more or less consciously by the OM users when using factories and creating instances of in-built classes, and by Lisp developers when defining classes and methods, there are very few cases of composers actually using object-orientation in the visual language as a tool for reaching some particular objective or reflecting formal compositional concepts. A noteworthy exception is the use of classes and hierarchy in OMChroma [4], a system devised by composer Marco Stroppa for the control of sound synthesis where classes correspond to underlying synthesis modules and are conceptually related to "sound potentials" or "equivalence classes" of sounds (see Figure 20). From these classes, subclasses can be graphically created or personalized, and then instantiated in OM patches with parametric values in order to generate concrete sounds.

For a more complete overview of the application of OM in music composition, the reader can refer to *The OM Composer's Book* series [5, 12]: Each chapter in these books is written by a composer describing his/her use of the visual programming language in relation to a particular musical approach and for the creation of some specific works.

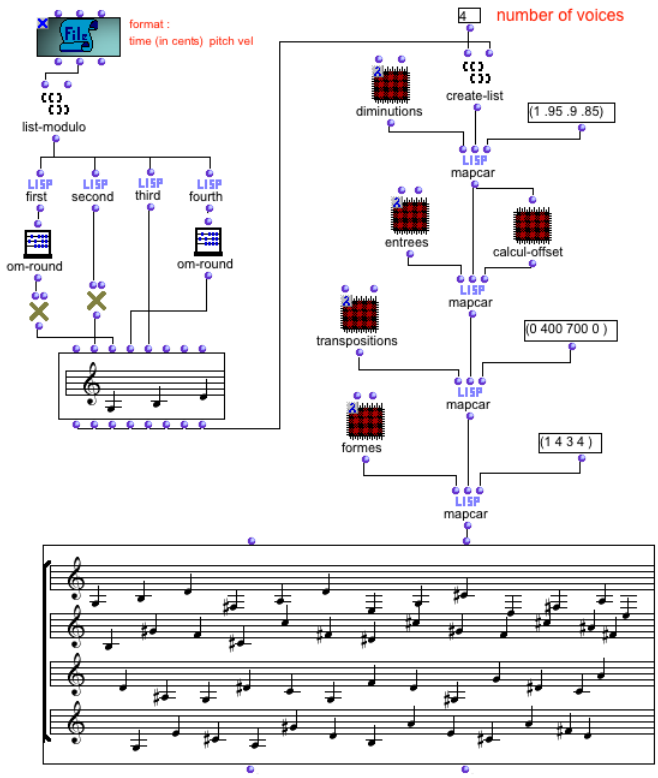

FIGURE 19: *Patch by Asbjørn Schaathun and Serge Lemouton designed for the composition of the piece Double Portrait (2006) [47]. Note the patch boxes (abstractions) in "lambda" mode, allowing to process and transform input data by successive higher-order function calls.*

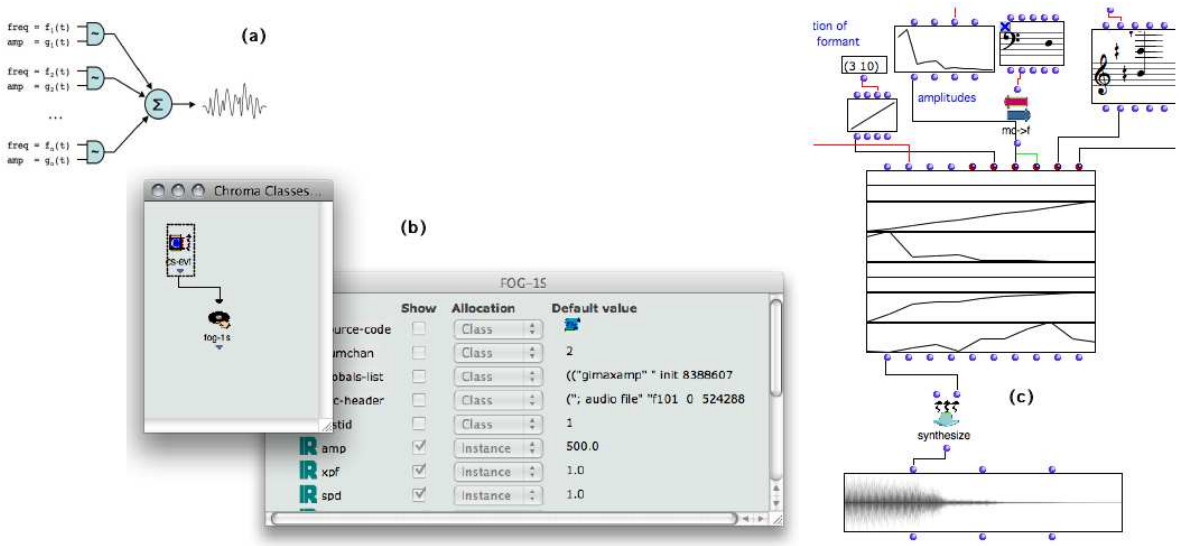

FIGURE 20: *An example of application of the OM object-oriented paradigm in a compositional approach: OMChroma. a) A signal processing module, with a given set of inputs; b) creation of a class in OM with corresponding slots; c) instantiation of the class in a visual program.*

13 Conclusion

We have presented some aspects of the OM visual programming language, and more specifically of its functional and object-oriented programming features. As we tried to demonstrate, almost all of the main programming techniques and constructs commonly used in Lisp/CLOS (abstraction, recursion, control structures, classes/methods definitions etc.) can be implemented graphically. OM fully supports functional and object-oriented paradigms, and both imperative and declarative approaches (several works on constraints programming have given fruitful results [8, 68]). Through a close similarity and an extensive use of the MOP in the language design, access is provided to most underlying programming structures and to the native components of the language itself.

The diagram model is not strictly applied and replaced by other visualization modes as soon as they appear more convenient: Documents browsing, for example, is done through desktop file/folder representations; and data setting protocols generally make use of form-like interfaces, or other advanced and specialized editors. Moreover, the visual tools can be completed at every level by text programming. The integration of textual and visual parts, along with abstraction features and the implementation of persistent objects and processes, provide effective means to address the program scalability issue. According to the usual VPL evaluation criteria OM can therefore be considered a fairly complete language [38], and overcomes many weaknesses identified in early visual programming languages (e.g. lack of formal specification, problems of large-scale program design, text/graphical integration, comments and secondary notation, etc. [54]).

Further works and improvements on the OM visual language could however be envisaged, for instance concerning the object-oriented framework: Improving its usability and interface would perhaps increase composer's interest and implication in this paradigm, which keeps being harder to understand and explore for composers (see Section 12). As mentioned above, simultaneous text/visual representation of a same program, function or class is theoretically partially possible and could also be an interesting approach to the integration of these two programming modalities.

Also note that the *macros*, which count in the most powerful features of Common Lisp, are not implemented graphically in OM and thereby constitute one of its major lacks concerning the visual/Lisp programming symmetry. A visual macro could be imagined as a patch generating an other patch when evaluated: This is not straightforward but theoretically possible, though probably of a little use for OM users. Finally, in a more prospective direction, the relevance and implementation of different interactive program execution modes could be investigated as an evolution of the language, as well as the possibility to provide efficient visual debugging tools.

Complementarily to the general-purpose programming features, OM provides an important set of classes, data structures and predefined functions allowing to develop musical processes. We shall also mention the existence of other types of programming interfaces, which have not been described here and allow to create musical structures in the form of programs extended with temporal semantics (*maquettes*, see [7, 1]), or to integrate programs in the context of score documents [11].

Generally speaking, and whatever it might be designed for, a crucial criteria for a visual programing environment is its ability to support the creation of real-life works and projects. In this matter, and probably also because it was designed in a strong "application-oriented" perspective, OM has proven to be successful and counts central roles in the production of numerous contemporary musical pieces created during the past 10 years. Although we mainly presented it as a general programming environment, the actual audience of OpenMusic is principally a community of contemporary music composers with very heterogeneous programming skills and general familiarity with the computer tools. After years of collaborations and pedagogical support, it seems like music creation became a domain where visual programming effectively succeeded in being adopted. This emphasizes another possible role of visual programming as a pedagogical tool, likely to introduce programming to specific communities of computer users, and to progressively change the way to conceive the relation between the user and the system in such creative applications.

14 Acknowledgements

An initial draft of this paper has been presented in the work-in-progress session at the 1st European Lisp Symposium – ELS'08 in Bordeaux, France, 2008. The authors would like to thank the organizers and participants of this session, as well as the successive reviewers of this paper for HOSC who helped a great deal in improving it and bringing it to its present form.

OpenMusic is developed at IRCAM in the Music Representations research group. The sources are available under GNU GPL license, and can be found online with other resources and information.[6] OM is also distributed with additional external libraries and as a compiled application running on MacOSX and Windows via the IRCAM forum.[7]

References

[1] Agon, C. *OpenMusic : Un langage visuel pour la composition musicale assistée par ordinateur*. PhD Thesis, Université Pierre et Marie Curie (Paris 6), 1998.

[2] Agon, C. and Assayag, G. "Programmation visuelle et éditeurs musicaux pour la composition assistée par ordinateur", *14ème Conférence Francophone sur l'Interaction Homme-Machine IHM'02*, Poitiers, France, 2002.

[3] Agon, C. and Assayag, G. "OM: A Graphical extension of CLOS using the MOP", *Proceedings of the International Lisp Conference – ILC'03*, New York, USA, 2003.

[4] Agon, C., Stroppa, M. and Assayag, G. "High Level Musical Control of Sound Synthesis in OpenMusic" *Proc. International Computer Music Conference*, Berlin, Germany, 2000.

[5] Agon, C., Bresson, J. and Assayag, G. (eds.) *The OM Composer's Book*, Vol. 1, IRCAM – Editions Delatour France, 2006.

[6] Assayag, G. "Computer Assisted Composition today", *1st symposium on music and computers*, Corfu, Greece, 1998.

[7] Assayag, G., Rueda, C., Laurson, M., Agon, C. and Delerue, O. "Computer Assisted Composition at IRCAM: From PatchWork to OpenMusic", *Computer Music Journal*, 23(3), 1999.

[8] Bonnet, A and Rueda, C. "Situation: Un langage visuel basé sur les contraintes pour la composition musicale", *in Recherches et applications en informatique musicale*, Chemillier M. and Pachet, F. (eds.), Hermes, 1998.

[9] Boshernitsan, M. and Downes, M. S. "Visual Programming Languages: A Survey", Technical Report CSD-04-1368, University of California, Berkeley, USA, 1997.

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

[11] Bresson, J. and Agon, C. "Programs, Scores, and Time Representations: The Sheet Object in OpenMusic", *Computer Music Journal*, 32(4), 2008.

[12] Bresson, J., Agon, C. and Assayag, G. (eds.) *The OM Composer's Book*, Vol. 2, IRCAM – Editions Delatour France, 2008.

[13] Burnett, M. "Visual Programming", In J. Webster, (ed.), *Encyclopedia of Electrical and Electronics Engineering*, John Wiley & Sons, 1999.

[6] http://repmus.ircam.fr/openmusic/
[7] http://forumnet.ircam.fr/

[14] Burnett, M. "Software engineering for Visual Programming Languages", *Handbook of Software Engineering and Knowledge Engineering*, Vol. 2, World Scientific Publishing Company, 2001.

[15] Burnett, M. M. and Baker, M. J. "A Classification System for Visual Programming Languages", *Journal of Visual Languages and Computing*, 5(3), 1994, pp. 287-300.

[16] Burnett, M., Atwood, J., Walpole Djang, R., Reichwein, J., Gottfried, H. and Yang, S. "Forms/3: A first-order visual language to explore the boundaries of the spreadsheet paradigm", *Journal of Functional Programming*, 11(2), Cambridge University Press, New York, NY, USA, 2001, pp. 155-206.

[17] Citrin, W., Doherty, M. and Zorm, B. "Formal Semantics of Control in a Completely Visual Programming Language", *IEEE Symposium on Visual Languages*, St Louis, USA, 1994, pp. 208-215.

[18] Cox, P., Giles, F. and Pietrzykowski, T. "Prograph: A step towards liberating programming from textual conditioning", *IEEE Workshop on Visual Languages*, Rome, Italy, 1989.

[19] Cox, P. T. and Pietryzkowsky, T. "Using a pictorial representation to combine data flow and object-orientation in a language-independent programming mechanism." In Glinert, E. P. (ed.), *Visual Programming Environments: Paradigms and Systems*, IEEE Computer Society Press, 1990.

[20] Dannenberg, R. B., McAvinney, P. and Rubine, D. "Arctic : A Functional Language for Real-Time Systems", *Computer Music Journal*, 10(4), 1986.

[21] Dannenberg, R. B. "Machine Tongues XIX: Nyquist, a Language for Composition and Sound Synthesis", *Computer Music Journal*, 21(3), 1997.

[22] Dannenberg, R. B., Desain, P. and Honing, H. "Programming Language Design for Music", *in* Roads. et al. (eds.), Musical Signal Processing, Swets and Zeitlinger, 1997.

[23] Eckel, G. and Gonzalez-Arroyo, R. "Musically Salient Control Abstractions for Sound Synthesis", *in Proceedings of the International Computer Music Conference*, Aarhus, Denmark, 1994.

[24] Erwig, M. and Meyer, B. "Heterogeneous Visual Languages – Integrating Visual and Textual Programming", *Proc. IEEE Symposium on Visual Languages*, Darmstadt, Germany, 1995, pp. 318–325.

[25] Fukunaga, A. S., Kimura, T. D. and Pree, W. "Object-Oriented Development of a Data Flow Visual Language System", *IEEE Symposium on Visual Languages*, Bergen, Norway, 1993, pp. 134-141.

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

[27] Glinert, E. P. and Tanimoto, S. L. "PICT: An Interactive, Graphical Programming Environment", *IEEE Computer*, 17, 1984, pp. 7-25.

[28] Glinert, E. P., Kopache, M. E. and McIntyre, D. W. "Exploring the General-Purpose Visual Alternative", *Journal of Visual Languages and Computing*, 1(1), 1990, pp. 3-39.

[29] Green, T. R. G. and Petre, M. "When visual programs are harder to read than textual programs", *Proceedings of the Sixth European Conference on Cognitive Ergonomics (ECCE 6)*, 1992, pp. 167-180.

[30] Green, T. R. G. and Petre, M. "Usability Analysis of Visual Programming Environments: A 'Cognitive Dimensions' Framework", *Journal of Visual Languages and Computing)*, 7(2), 1996, pp. 131-174.

[31] Haddad, K. "Livre Premier de Motets: The Time-Block Concept in OpenMusic." In Bresson, J., Agon, C. and Assayag, G. (eds.) *The OM Composer's Book*, Vol. 2, IRCAM – Editions Delatour France, 2008.

[32] Helsel, R. *Visual programming with HP VEE (2nd ed.)*, Prentice-Hall, Inc., Upper Saddle River, USA, 1997.

[33] Hiller, L. *Revised MUSICOMP manual*, University of Illinois Experimental Music Studio, USA, 1969.

[34] Hils, D. D. "DataVis: A Visual Programming Language for Scientific Visualization", *Proceedings of the ACM Computer Science Conference*, San Antonio, USA, 1991.

[35] Hils, D. D. "Visual Languages and Computing Survey: Data Flow Visual Programming Languages", *Journal of Visual Languages and Computing*, 3(1), 1992.

[36] Holt, C. M. Viz: A Visual Language Based on Functions, *IEEE Workshop on Visual Languages*, Skokie, Illinois, 1990.

[37] Kiczales, G., des Rivières, J. and Bobrow, D. G. *The Art of the Metaobject Protocol*, MIT Press, 1991.

[38] Kiper, J. D., Howard, E. and Ames, C. "Criteria for Evaluation of Visual Programming Languages", Journal of Visual Languages and Computing, 8(2), 1997, pp. 175-192.

[39] Kimura, T. D., Choi, J. W. and Mack, J. M. *A Visual Language for Keyboardless Programming*, Technical Report, WUCS-86-6, Univesity of Washington, St Louis, USA, 1986.

[40] Kuuskankare, M. et Laurson, M. "Expressive Notation Package", *Computer Music Journal*, 30(4), 2006.

[41] Larkin, J. H. and Simon, H. A. "Why a Diagram is (Sometimes) Worth Ten Thousand Words", *Cognitive Science*, 11, 1987, pp. 65-99.

[42] Lau-Kee, D., Billyard, A., Faichney, R., Kozato, Y., Otto, P., Smith, M. an Wilkinson, I. "VPL: An Active, Declarative Visual Programming System", *IEEE Workshop on Visual Languages*, Kobe, Japan, 1991, pp 40-46.

[43] Laurson, M. "PWConstraints", *Symposium: Composition, Modélisation et Ordinateur*, IRCAM, Paris, 1996.

[44] Laurson, M. and Duthen, J. "Patchwork, a Graphic Language in PreForm", *Proceedings of the International Computer Music Conference*, Ohio State University, USA, 1989.

[45] Laurson, M. and Kuuskankare, M. "PWGL, a Novel Visual Language Based on Common Lisp, CLOS, and OpenGL", *Proceedings of the International Computer Music Conference*, Gothenburg, Sweden, 2002.

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

[47] Lemouton, S. and Schaathun, A. "Knitting and Weaving: Using OpenMusic to Generate Canonic Musical Material." In Bresson, J., Agon, C. and Assayag, G. (eds.) *The OM Composer's Book*, Vol. 2, IRCAM – Editions Delatour France, 2008.

[48] Letz, S., Orlarey, Y. and Fober, D. "The Role of Lambda-Abstraction in Elody", *Proceedings of the International Computer Music Conference*, Ann Arbor, Michigan, USA, 1998.

[49] Malt, M. *Les mathématiques et la composition assistée par ordinateur (concepts, outils et modèles)*, PhD. Thesis, Ecoles des Hautes Etudes en Sciences Sociales, Paris, France, 2000.

[50] Mathews, M. *The Technology of Computer Music*, MIT Press, 1969.

[51] McCartney, J. "Rethinking the Computer Music Language: SuperCollider", *Computer Music Journal*, 26(4), 1996.

[52] Melen, C. "Six Metal Fugue." In Bresson, J., Agon, C. and Assayag, G. (eds.) *The OM Composer's Book*, Vol. 2, IRCAM – Editions Delatour France, 2008.

[53] Monden, N., Yoshimoto, I., Hirakawa, M., Tanaka, M. and Ichikawa, T. "HI-VISUAL: A language supporting visual interaction in programming", *IEEE Workshop on Visual Languages*, 1984.

[54] Myers, B. A. "Taxonomies of Visual Programming and Program Visualization", *Journal of Visual Languages and Computing*, 1(1), 1990.

[55] Najork, M. A. "Programming in Three Dimensions", *Journal of Visual Languages and Computing*, 7(2), 1996.

[56] Najork, M. A. and Golin, E. "Enhancing Show-And-Tell with a Polymorphic Type System and Higher-Order Functions", In *IEEE Workshop on Visual Languages*, 1990.

[57] Orlarey, Y., Fober, D. and Letz, S. "Elody : a Java+MidiShare based Music Composition Environment", *Proceedings of the International Computer Music Conference*, Thessaloniki, Greece, 1997.

[58] Petre, M. and Green, T. R. G. "Learning to Read Graphics: Some evidences that 'Seeing' an Information Display is an Acquired Skill", *Journal of Visual Languages and Computing*, 4(1), 1993.

[59] Pope, S. T. (ed.) *The Well Tempered Object (Musical Applications of Object-Oriented Software Technology)*, MIT Press, 1991.

[60] Poswig, J., Vrankar, G. and Morara, C. "VisaVis: a Higher-order Functional Visual Programming Language", *Journal of Visual Languages and Computing*, 5(1), 1994.

[61] Puckette, M. "Combining Event and Signal Processing in the MAX Graphical Programming Environment", *Computer Music Journal*, 15(3), 1991.

[62] Rasure, J. R. williams, C. S. "An Integrated Data Flow Visual Language and software Development Environment", *Journal of Visual Languages and Computing*, 2(3), 1991.

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

[64] Rueda, C., Alvarez, G., Quesada, L. O., Tamura, G., Valencia, F. D., Díaz, J. F. and Assayag, G. "Integrating Constraints and Concurrent Objects in Musical Applications: A Calculus and its Visual Language", *Constraints*, 6(1), 2001.

[65] Schiffer, S. and Fröhlich, J. H. "Visual Programming and Software Engineering with Vista", *In* Burnett, M., Goldberg, A. and Lewis, T. G. (eds.) *Visual Object-Oriented Programming: Concepts and Environments*, Manning Publications Co., Greenwich, USA, 1995.

[66] Steele, G. L. *Common Lisp the Language*, 2nd Edition, Digital Press, 1990.

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

[68] Truchet, C., Assayag, G. and Codognet, Ph. "OMClouds, a heuristic solver for musical constraints", *MIC'03 Metaheuristics International Conference*, Kyoto, Japan, 2003.

[69] Tyugu, E. and Valt, R. "Visual programming in NUT", *Journal of Visual Languages and Computing*, 8(5-6), 1997.

[70] Vose, G. M. and Williams, G. "LabVIEW: Laboratory virtual instrument engineering workbench", *Byte*, 11, 1986, pp. 84-92.

[71] Whitley, K. N. "Visual Programming Languages and the Empirical Evidence For and Against", *Journal of Visual Languages and Computing*, 8(1), 1997, pp. 109-142.

[72] Whitley, K. N. and Blackwell A. F. "Visual Programming: The Outlook from Academia and Industry", *Proceedings of the 7th Workshop on Empirical Studies of Programmers*, Alexandria, USA, 1997.

[73] Young, M., Argiro, D., and Kubica, S. "Cantata: Visual Programming Environment for the Khoros System", *Computer Graphics*, 29, 1995.